AMSTRAD

TURBO PASCAL

DOUGLAS S. STIVISON



AMSTRAD
TURBO PASCAL

Ouvrages sur le Pascal parus aux éditions Sybex

Introduction au Pascal, de P. Le Beux
Le guide du Pascal, de J. Tiberghien
Le Pascal par la pratique, de P. Lebeux et H. Tavernier
Programmes en Pascal, de A. R. Miller
Introduction à Mac Pascal, de P. Le Beux
Jeux en Pascal sur Apple, de D. Hergert et J.T. Kalash

AMSTRAD TURBO PASCAL

Douglas S. Stivison



Paris • Berkeley • Düsseldorf

Turbo Pascal est une marque de Borland International Inc.
BASICA et MS sont des marques déposées de Microsoft Corp.
CompuServe est une marque déposée de CompuServe, Inc.
CP/M, CP/M-80, CP/M-86 et Pascal MT+ sont des marques déposées de Digital Research, Inc.
IBM est une marque déposée d'International Business Machines Corp.
Radio Shack et Model 4 sont des marques de Tandy Corp.
UCSD p-System est une marque déposée de Regents of the University of California.
WordStar est une marque déposée de MicroPro International Corp.
XyWrite est une marque de XyQuest, Inc.

Tous les efforts ont été faits pour fournir dans ce livre une information complète et exacte. Néanmoins, SYBEX n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

ISBN 2-7361-0223-1

ISBN 0-89588-269-8 version originale.

SOMMAIRE

INTRODUCTION

Ce livre a été écrit pour aider les utilisateurs à développer aussi rapidement que possible des programmes en Turbo Pascal. Il doit également aider le lecteur à comprendre la programmation structurée.

Les livres existant sur le Pascal ne rendent pas compte des caractéristiques uniques du Turbo. Ses extensions et ses variations par rapport au standard ne sont pas uniquement des modifications mineures. Dans le Turbo Pascal, Borland a greffé une vaste gamme d'extensions et de simplifications sur la structure de base du Pascal mis au point par Niklaus Wirth. Ce langage a été initialement développé dans un environnement universitaire pour être utilisé sur de gros ordinateurs. La puissance des micro-ordinateurs personnels a permis de passer outre les contraintes des Pascal antérieurs et d'envisager les innovations du Turbo. Ce guide relatif au Turbo a été écrit pour qu'un nombre croissant d'utilisateurs puissent bénéficier de ses exceptionnelles capacités.

Celles-ci sont trop puissantes pour être ignorées, trop élégantes pour être enterrées dans une documentation obscure. De plus, ce mélange de caractéristiques standard, d'extensions universelles et d'options dépendantes du matériel employé est un triomphe de cohérence et de convivialité. D'autres Pascal exploitables sur des micro-ordinateurs ont la mauvaise réputation d'être d'une mise en œuvre très délicate. Ce reproche n'est absolument pas applicable au Turbo et nous espérons que ce livre montrera qu'il est très facile à maîtriser.

Le Turbo est un langage très accessible et cet ouvrage est un guide destiné à mettre l'utilisateur en situation.

Pour qui ce livre a-t-il été écrit ?

Ce livre est destiné aux personnes désireuses de faire de la programmation, aux gestionnaires cherchant un outil pour résoudre des problèmes courants, aux programmeurs chevronnés ainsi qu'aux étudiants, qui peuvent accéder au Turbo étant donné son faible prix.

Ce livre est également destiné à ceux qui ont été récemment attirés par l'informatique et qui sentent qu'ils ont déjà atteint les limites du BASIC interprété. Le Turbo propose à ces lecteurs des outils leur permettant de manipuler des fichiers importants, de générer un graphisme plus rapide et de résoudre des problèmes plus ambitieux.

Organisation de ce livre

Le Chapitre 1 propose une brève approche de la philosophie du Turbo Pascal en révélant sa structure de base. Le Pascal est un langage exceptionnellement précis et logique. Sa syntaxe et sa structure ne constituent pas un système de règles intimidantes, ni une construction et une ponctuation arbitraires, mais forment un ensemble cohérent. Lors de l'apprentissage d'un langage naturel, nous faisons rapidement des suppositions pour exprimer des idées complexes à partir de modèles simples. De même, lors de l'écriture de programmes Turbo avancés, il existe de nombreux codes de contrôle et structures de données qui sont basés sur le fonctionnement de structures simples.

Le Chapitre 2 se sert de programmes simples pour familiariser le lecteur avec la structure élémentaire d'un programme en Pascal. Pour les utilisateurs de langages non structurés tels que le BASIC ou le FORTRAN, un programme en Pascal peut sembler déroutant. Nous espérons que ces quelques exemples opérationnels rendront la transition en format Pascal intuitive et non mécanique. Ce chapitre couvre également la terminologie nécessaire aux utilisateurs pour commencer à écrire, à compiler et à faire tourner des petits programmes personnels.

Le Chapitre 3 traite des procédures, fonctions, structures de contrôle et structures de données courantes qui constituent presque tous les programmes du Turbo. Ce chapitre présente également les capacités qui font la force du Turbo : manipulation de chaînes et fonctions créées par l'utilisateur.

Le Chapitre 4 concerne exclusivement les structures de données avancées y compris les enregistrements et les fichiers. Il est universellement reconnu que la principale qualité du Turbo consiste en son jeu de structures de données très souple qui reflète les relations "naturelles" établies entre les données. En plus de ces techniques destinées à utiliser des fichiers de données externes, on trouve dans ce chapitre la gamme de structures de données allant des tableaux jusqu'aux listes utilisant les pointeurs. C'est un sujet auquel la plupart des utilisateurs du Pascal reviennent sans cesse au fur et à mesure que leur expérience grandit.

Le Chapitre 5 aborde les commandes du Turbo permettant de générer du graphisme. Il propose les outils appropriés pour rendre les programmes plus agréables. Les programmeurs Turbo débutants découvriront ici des applications nouvelles.

Le Chapitre 6 traite de sujets intéressant les programmeurs chevronnés. Ceux-ci incluent le fonctionnement du compilateur Turbo et les différentes manières d'accéder aux fonctions offertes par le système d'exploitation. Ce Chapitre explique et compare aussi les différentes techniques de gestion et de chaînage de programmes importants.

Enfin, l'Annexe propose le listing d'un programme de gestion d'une liste d'adresses dont certaines parties apparaissent sous une forme différente dans les chapitres précédents.

Les connaissances requises

Le lecteur est supposé avoir une expérience d'un autre langage de programmation tel que le FORTRAN ou le BASiC. Aucune connaissance d'un langage structuré quelconque n'est indispensable. Le lecteur doit se référer au manuel Borland pour tout ce qui concerne l'installation et l'initialisation du logiciel Turbo sur son propre système.

Et maintenant, lecteurs et lectrices, à vos disquettes !

1. LES ÉLÉMENTS FONDAMENTAUX DU TURBO PASCAL

PRÉSENTATION

Le Turbo Pascal n'est pas seulement un jeu de commandes différent pour exprimer les mêmes idées que le BASIC. Comme les autres Pascal, il représente une approche fondamentalement différente de la programmation et même un mode de pensée particulier. Si on passe quelque temps à explorer la philosophie qui le guide, l'apprentissage des détails du langage sera plus facile car tous ses éléments correspondent à une trame logique.

L'utilisation du Turbo nécessite un processus de pensée structuré ; le signe le plus évident de cette discipline est la structure de blocs ordonnés. Par contraste, un langage tel que le BASIC a une structure amorphe. Les programmes Turbo sont écrits bloc par bloc, chaque bloc étant destiné à un but spécifique clairement défini. Les commandes, la syntaxe, les tests, les boucles et la présentation du programme reflètent cette orientation fondamentale consistant à fragmenter des opérations complexes en unités facilement manipulables.

Niklaus Wirth, concepteur du Pascal et de Modula 2, a une conviction fondamentale : une programmation efficace consiste à résoudre deux problèmes distincts mais associés. Le premier consiste à définir les *structures de données* (quel type et quelle quantité d'information doivent être traités ? Quelle est la manière la plus naturelle, la plus pratique et la plus efficace de les gérer ?).

Le second problème concerne la création d'un *algorithme* : quelle est la meilleure procédure pour générer le résultat désiré à partir de ces données ? Ce cheminement explique la logique qui sous-tend les commandes et la syntaxe du Turbo ainsi que ses outils de structuration de données qui offrent des possibilités bien supérieures à celles des langages précédents. Les programmeurs Pascal débutants considèrent souvent ces nouveaux types de données et ces structures comme des complications du concept familier de tableaux du BASIC. En réalité, il s'agit d'outils très puissants permettant de décrire des situations réelles en des termes compréhensibles par l'ordinateur.

Les programmeurs Turbo développent rapidement leurs capacités à créer des algorithmes en pensant à des *processus* traitant des structures de données clairement définies. Par contre, la plupart des programmeurs BASIC ont du mal à visualiser la structure d'un programme à partir d'une représentation ligne par ligne.

Une fois l'orientation du Turbo comprise et assimilée, sa syntaxe et sa ponctuation ne sont plus des règles arbitraires à apprendre par cœur mais constituent la manière la plus naturelle d'exprimer des processus clairement définis.

Au cours de cet ouvrage, on insistera sur les deux principes suivants : les programmes sont organisés en blocs et les blocs sont des algorithmes qui travaillent sur des structures de données.

PROGRAMME DE DÉMONSTRATION

Avant d'entrer dans les détails de la création d'un programme, nous allons examiner un exemple :

```
PROGRAM Affichage_de_Salutations ;
BEGIN
      WRITELN ('Bonjour les petits amis') ;
END
```

Ce programme sans prétention affiche simplement sur l'écran le message :

Bonjour les petits amis

Aussi simple soit-il, il illustre quelques principes fondamentaux du Turbo. Tout d'abord, les programmes Turbo (même ceux qui

ne sont pas "documentés " par des commentaires) doivent indiquer leur but. Pour cela, le Turbo laisse une grande liberté au programmeur dans l'attribution de noms aux éléments du programme et dans la présentation physique des lignes de code sur l'écran ou sur papier. C'est à l'utilisateur de tirer parti de cette liberté en employant des noms significatifs pour décrire ce que fait une procédure ou ce que représente une variable et en indentant les lignes ou en ajoutant des lignes vierges pour rendre la structure logique évidente à n'importe quel utilisateur.

Fragment de programme

Ce programme très simple est constitué de deux modules ou blocs. La première partie :

```
PROGRAM Affichage_de_Salutations ;
```

est appelée *en-tête* du programme. C'est le nom du programme qui indique éventuellement où trouver les entrées et les sorties. A l'inverse du Pascal standard, le Turbo rend cette partie optionnelle. Cependant, il est recommandé de nommer un programme car son titre renseigne l'utilisateur sur sa fonction. Dans cet exemple, il est évident que le programme a pour but d'afficher un message de salutation.

La seconde partie est le *corps* du programme ; elle contient le code qui accomplit effectivement la tâche et n'est évidemment pas optionnelle. Dans notre exemple, le corps du programme est :

```
BEGIN
        WRITELN ('Bonjour les petits amis') ;
END
```

Cet exemple est caractéristique, car une donnée (le message) est traitée (affichée sur l'écran). On remarque que certains mots apparaissent en majuscules ; ce sont les mots clés du Turbo. Sans connaître ces commandes, on voit, grâce à l'identation et aux majuscules des mots PROGRAM, BEGIN et END, l'objet du programme (affichage de salutations). Les programmeurs BASIC remarqueront également qu'il n'y a pas de numéros de lignes ; ceux-ci ne sont pas nécessaires lorsqu'un programme est organisé en blocs nommés de façon significative avec un début et une fin sans ambiguïté.

La disquette du logiciel comprend un programme CALC.PAS. Il s'agit d'une feuille de calcul ambitieuse, composée de nombreux modules et capable d'effectuer une quantité phénoménale de calculs et de manipulations d'écran.

Bien que cet ouvrage n'examine pas le programme CALC.PAS en détail, on peut le regarder pour voir la manière dont il est structuré. La liste des variables et les noms des procédures donnent immédiatement une idée de la tâche qu'il accomplit.

Installation du Turbo

La documentation Borland explique comment installer le Turbo sur l'ordinateur employé et comment utiliser et adapter l'éditeur de programme associé. Il faut réaliser une copie de sauvegarde du logiciel pour éviter sa perte en cas d'accident.

L'environnement Turbo

Le Turbo n'est pas seulement un langage de compilation, mais il forme un environnement de développement de programmes totalement intégré. Les autres Pascal, comme tous les langages de haut niveau, nécessitent un va-et-vient entre le système d'exploitation, le traitement de texte, un compilateur à une, deux ou trois passes et un éditeur de liens. La modification d'un programme simple prend du temps et est source d'erreurs.

La frappe de la commande TURBO, à partir du système d'exploitation, permet d'accéder à un menu à partir duquel on peut créer et éditer des programmes, réaliser des compilations et des éditions de liens, tester et mettre au point des programmes et enfin générer un programme exécutable en dehors de l'environnement Turbo. Seuls les programmeurs qui ont mis au point des programmes dans d'autres langages ou dans d'autres versions du Pascal peuvent apprécier les avantages offerts par cette intégration.

Voici un scénario de développement de programme Turbo classique. Au signal du système d'exploitation, l'utilisateur tape :

TURBO

L'éditeur Turbo est ensuite sélectionné en tapant la touche E et un fichier source est rapidement affiché pour l'édition. Les modifications sont effectuées et avec deux frappes de touches sup-

plémentaires, le programme est immédiatement recompilé et exécuté. Lorsque le compilateur trouve des erreurs, il place des messages appropriés dans le fichier source ; après correction, le programme peut être à nouveau compilé, exécuté et testé. Ce cycle est répété jusqu'à la mise au point complète du programme. A ce stade, toujours à l'intérieur du Turbo, il est possible de créer un fichier .COM exécutable en dehors du Turbo.

En programmation normale, le cycle classique d'édition, de compilation, d'édition de liens et d'exécution doit être répété plusieurs fois. Pour un programme d'une trentaine de lignes, les compilateurs Pascal passent trois à dix minutes par cycle alors que le Turbo n'a besoin que de quelques secondes ; la différence est encore plus marquée avec des programmes plus complexes. On imagine la frustration engendrée par une attente de vingt minutes pour s'apercevoir qu'un point-virgule a été oublié. Cela est courant dans de nombreux Pascal, mais absolument inconnu en Turbo.

La principale raison de la rapidité du Turbo est due au fait que la plupart des programmes sont édités, entièrement compilés, exécutés et mis au point dans la mémoire centrale de l'ordinateur. Cette méthode est beaucoup plus rapide que les approches classiques qui nécessitent du temps pour charger successivement l'éditeur, les fichiers programme, les bibliothèques, les compilateurs et l'éditeur de liens en mémoire centrale à partir du disque.

La méthode du Turbo comporte quelques inconvénients ; par exemple, il n'y a pas d'élaboration automatique de fichiers listing ou de listings de code objet. Cependant, ces éléments n'intéressent généralement que les programmeurs professionnels. Le programme TLIST affiche les listings dont la plupart des utilisateurs ont besoin ; les professionnels peuvent toujours utiliser le Turbo pour écrire un éditeur de code objet si nécessaire.

2. PROGRAMMATION ÉLÉMENTAIRE EN TURBO PASCAL

CRÉATION ET EXÉCUTION D'UN PREMIER PROGRAMME

Il est temps maintenant de se lancer dans la création d'un programme en Turbo Pascal. Il faut s'assurer auparavant que le Turbo a bien été installé suivant les spécifications du manuel. On tape :

TURBO

après l'indicateur du système d'exploitation. Le logiciel est chargé instantanément et propose à l'utilisateur le chargement du fichier de messages d'erreur :

Include error messages (Y/N) ?

La plupart des compilateurs Pascal n'offrent pas cette possibilité ; à la place, il n'y a que le manuel et, en cas d'erreur, le programmeur doit, dans un dédale de messages cryptés, rechercher les explications du code d'erreur. Si on tape Y (oui), le logiciel affiche un descriptif associé au type de l'erreur commise. Sur des systèmes ne possédant qu'une taille mémoire restreinte, le fichier des messages d'erreur (TURBO.MSG) ne doit pas être chargé pour réserver un espace plus grand au programme ; cependant, il est possible de le conserver sur disque. Les messages inclus dans le programme sont très utiles à l'utilisateur lors de l'élaboration de programmes.

L'écran affiche ensuite le menu principal du Turbo. Les commandes sont choisies en tapant la touche correspondant à la première lettre de chaque fonction ; on remarque que celle-ci apparaît en surbrillance. Pour écrire un nouveau programme, on tape W (de manière à nommer un fichier de travail : *Workfile*). Son nom est ensuite défini :

Workfile name : PREMIER

Un message précise que PREMIER.PAS n'existe pas encore et qu'il s'agit donc d'un nouveau fichier :

New File

Si nous avions choisi la lettre E (*Edit*), il aurait fallu préciser le nom d'un programme existant déjà comme avec n'importe quel autre logiciel de traitement de texte. Comme pour n'importe quel langage, les programmes en Turbo ne sont pas issus d'une génération spontanée ; il est nécessaire de les écrire...

On tape l'exemple de programme du Chapitre 1 :

```
PROGRAM Affichage_de_salutations ;
BEGIN
    WRITELN ('Bonjour les petits amis')
END.
```

L'éditeur du Turbo accepte indifféremment les combinaisons de majuscules et de minuscules. Cependant, dans certains cas, écriture dans un fichier ou bien sortie sur un périphérique (écran ou imprimante par exemple), il faut choisir entre les deux.

Les espaces et les indentations dont nous avons déjà parlé dans le Chapitre 1 ne dépendent que de l'humeur du programmeur. Mais, si aucune règle n'est établie, il suffit de se mettre à la place d'un utilisateur futur qui un jour devra lire le programme. L'éditeur du Turbo offre à l'utilisateur la possibilité d'indenter automatiquement les lignes de programme ; très rapidement, il est possible de jongler avec ces fonctions. Du point de vue de l'ordinateur, il aurait été possible d'écrire le programme de la manière suivante :

```
BEGIN WRITELN('Bonjour les petits amis')END.
```

Pour les utilisateurs habitués au BASIC, cette souplesse est une distinction majeure. En BASIC, il n'y a qu'un fichier programme

pour exécuter une ou plusieurs tâches ; il peut contenir des lignes de commentaire et des lignes vierges. Si on ajoute une série de remarques ou si on structure les opérations complexes sur plusieurs lignes (avec des commentaires), le programme devient beaucoup plus facile à lire ; hélas, il est plus long et souvent plus lent. Pour des petits programmes, la vitesse importe peu ; mais dans des programmes plus ambitieux, tris ou communications, la taille et donc la lenteur peuvent compliquer l'exécution. Ce sont précisément ces programmes longs qui requièrent le plus de commentaires et une structure claire afin que le concepteur ne soit pas le seul à pouvoir les comprendre.

Le Turbo Pascal est un langage compilé qui crée, à partir d'un fichier source, un fichier de commandes en langage machine qui permettra l'exécution du programme. C'est la taille et la structure de ce fichier en langage machine (parfois appelé fichier tâche et portant l'extension COM) qui déterminent l'occupation mémoire du programme et la nature de son déroulement. Lors de la création du fichier tâche par le compilateur Turbo, les commentaires, les lignes vides et les indentations ne sont pas pris en compte. De même les noms des variables, qui sont souvent longs afin d'évoquer leur contenu au programmeur, sont remplacés par des adresses mémoire. On remarque que les performances du programme ne sont pas altérées par l'adjonction de commentaires, de lignes blanches, d'une structure constituée d'indentations de blocs ou de noms de variables longs et significatifs.

Revenons maintenant à notre premier programme. L'édition se termine comme avec le traitement de texte WordStar par la frappe des caractères de contrôle ^KD. Le premier fichier source Turbo a été créé et sauvegardé. Pour le compiler et l'exécuter, on tape R (pour *Run*). Si le programme a été tapé correctement, le message :

```
Running
Bonjour les petits amis
>
```

apparaît sur l'écran (avec la version 3, l'écran n'est pas effacé).

SAUVEGARDE D'UN PROGRAMME

Nous avons écrit, compilé puis exécuté notre premier programme. Cependant, celui-ci n'existe que dans la mémoire centrale

de l'ordinateur ; il n'a pas été sauvegardé sur disque. Si l'ordinateur est mis hors tension, il n'existe plus aucune trace de ce programme. Lors de l'utilisation du Turbo, il est donc conseillé de faire une sauvegarde du fichier en cours sur disque. Si l'on sort du logiciel suivant la procédure normale et que la sauvegarde n'a pas été réalisée, le Turbo propose courtoisement de le faire :

Workfile A : \PREMIER.PAS not saved. Save (Y/N) ?

Ainsi, il n'est pas possible de quitter le programme et d'oublier de sauvegarder le fichier en cours. Les oublis les plus fréquents arrivent lorsque l'on passe alternativement du mode d'édition à l'exécution d'un programme test ; par exemple, on peut modifier un programme, le compiler, l'exécuter puis charger un nouveau programme en oubliant de sauvegarder la dernière mise à jour du précédent. Parfois aussi, le programme contient une erreur qui génère une boucle infinie, qu'il est impossible d'interrompre, même par la frappe d'une séquence Ctrl-Break. Il n'y a qu'une solution, recharger le système (*rebooter*) mais la dernière version du programme ne sera pas sauvegardée sur le disque et, au cours d'une séquence de mise au point, la dernière version stockée sur disque peut être très différente de la version en cours.

Il faut prendre l'habitude, avant chaque compilation, de sauvegarder le programme qui vient d'être édité en tapant la commande S (pour *Save*, sauvegarde).

EXÉCUTION D'UN PROGRAMME COMPILÉ AVEC LE SYSTÈME D'EXPLOITATION DE L'ORDINATEUR

Il est possible de modifier le programme en cours en tapant la commande E (Edit) et il ne faut pas oublier de le sauvegarder de temps en temps avec la commande S. Après écriture sur le disque, nous avons constaté que notre programme s'appelait PREMIER.PAS. Ce fichier ne peut être exécuté que dans l'environnement Turbo, c'est-à-dire après chargement du logiciel. Pour créer une version directement exécutable à partir du système d'exploitation, on tape O (*compiler Options* : options du compilateur) puis C (compilation en fichier .COM) et enfin Q (*Quit* pour quitter les options de compilation). La frappe de la touche C génère un fichier appelé PREMIER.COM. On sort ensuite du Turbo en tapant Q (Quit) ; le signal du système d'exploitation est affiché sur l'écran

(A>pour CP/M). On tape :

PREMIER

suivi de la frappe de la touche Return. Le message :

Bonjour les petits amis

est affiché sur l'écran.

Notre premier programme en Turbo Pascal vient d'être exécuté à partir du système d'exploitation, sans nécessiter la présence du Turbo sur le disque. Nous avons créé un fichier source, nous l'avons compilé, testé et mis au point dans l'environnement Turbo, puis recompilé afin de générer un programme directement exécutable. Voici encore une différence fondamentale entre le Turbo et les interpréteurs BASIC : si l'on se procure un programme écrit en BASIC (par exemple, BASIC Microsoft, BASIC Apple, BASIC Amstrad...), il faut s'assurer que l'interpréteur BASIC qui a servi à sa conception est identique à celui que nous possédons, sinon il est impossible de l'exploiter sans faire de modifications plus ou moins importantes.

Le Turbo génère des fichiers .COM standard exécutables sur tous les ordinateurs qui utilisent le même système d'exploitation ; par exemple, ordinateurs Amstrad. Il existe cependant des exceptions pour les programmes qui adressent des zones mémoire étendues ou qui font appel à certaines particularités du BIOS.

Cette portabilité fait du Turbo un langage particulièrement attractif pour les programmeurs qui souhaitent écrire et commercialiser du logiciel. Si le programmeur évite les commandes du Turbo les plus ésotériques, ses programmes seront exécutables sans modifications sur une large gamme de modèles d'ordinateurs : système 16-bits dont les systèmes d'exploitation sont MS-DOS ou CP/M-86 et systèmes 8-bits (basés sur le Z80) utilisant CP/M. Si le système d'exploitation est différent, les programmes source doivent être recompilés avec le compilateur Turbo spécifique de la machine.

Il faut noter que la portabilité du Turbo ne s'applique pas à tous les types de compilateurs Pascal (Microsoft, UCSD, Intel, Digital Research). Comme nous l'avons mentionné précédemment pour le BASIC, il est généralement délicat de convertir un programme source pour un autre compilateur Pascal.

Ce problème d'incompatibilité n'est pas seulement lié au Turbo ; historiquement, le Pascal a été imaginé comme un langage théo-

rique destiné à l'enseignement de la programmation structurée. Lorsqu'il fut installé sur des ordinateurs réels, il séduisit les utilisateurs par sa facilité et sa souplesse. Cependant, des lacunes importantes existaient : entrées/sorties et manipulations de chaînes de caractères. La plupart des compilateurs ont comblé cette lacune en s'éloignant plus ou moins du Pascal "standard " ; chaque version à ses adeptes qui lui préfèrent son jeu d'extensions, ses particularités par rapport au microprocesseur et ses compromis entre la théorie et la réalité.

Le Turbo Pascal s'éloigne du Pascal standard pour offrir à l'utilisateur des performances plus importantes et une mise en œuvre plus simple. Il aurait été possible dans ce livre de traiter le Turbo et ses différences avec le Pascal standard ; il semble plus intéressant de le considérer et de le présenter comme un langage indépendant.

Les algorithmes du Turbo, ses structures de données et ses modules peuvent généralement être utilisés sans problème pour générer un programme dans n'importe quel "dialecte " Pascal. Seuls les détails de syntaxe, les conventions de compilation et de linkage ainsi que la création de tâches présentent des différences radicales.

MOTS RÉSERVÉS, IDENTIFICATEURS ET IDENTIFICATEURS STANDARD

L'étape fondamentale de l'apprentissage du Turbo Pascal est la connaissance du vocabulaire de base puis la manière de combiner ses mots (instructions, commandes et modules) dans des structures significatives.

Le vocabulaire du Turbo Pascal est exceptionnellement simple ; il n'y a pas de symboles occultes comme en APL ni de ponctuation mystérieuse comme en C, mais plutôt, comme en BASIC, un vocabulaire imagé renseignant l'utilisateur sans ambiguïté. Les mots BEGIN et END qui limitent plusieurs lignes d'instructions mettent en évidence une zone particulière pour l'utilisateur, tandis qu'ils fournissent au compilateur un autre type de renseignement.

Les programmes en Turbo Pascal sont écrits avec une combinaison de mots réservés et d'identificateurs standard qui sont déjà connus du compilateur Turbo, puis avec des identificateurs qui sont créés et ajoutés par le programmeur. Comme dans n'importe quel langage naturel, il y a des règles de grammaire et de ponctuation

pour contrôler la combinaison des divers éléments. Cependant, le Pascal en général et le Turbo en particulier permettent une souplesse extraordinaire dans le choix du vocabulaire généré par l'utilisateur. Lors de la définition de données (variables et structures), de procédures ou de fonctions, le Turbo incite l'utilisateur à employer des noms longs et imagés. Nous allons maintenant revenir au concept "algorithmes + structure de données ".

Mots réservés

Les mots réservés sont des mots qui ont une signification particulière et immuable pour le compilateur Turbo ; ce sont les mots clés utilisés pour écrire un programme en Turbo Pascal. La Figure 2.1 présente la liste des mots clés du Turbo. Il n'est pas indispensable de connaître par cœur leur fonction car, en général, la signification (en anglais) des mots est étroitement liée à la tâche qu'ils imposent au compilateur ; de plus, leur connaissance se fera au cours de l'apprentissage du langage. Les mots réservés sont destinés à contrôler le déroulement logique et les opérations d'un programme. Ils comportent les termes nécessaires à l'exécution de tests et de boucles, et à l'accomplissement de calculs et de comparaisons. Le programmeur doit prendre garde à ne pas employer un mot réservé pour un nom de variable ou pour un nom de programme.

Dans la liste, on remarque des mots du Turbo qui n'existent pas dans le Pascal standard (ABSOLUTE, EXTERNAL, INLINE, SHL, SHR, XOR et STRING) ; les autres ont des fonctions pratiquement identiques dans les deux langages.

De la même manière, les autres Pascal possèdent des mots réservés qui leur sont propres et qui s'éloignent plus ou moins du Pascal standard et du Turbo. Le concept de mot réservé diffère également d'une implémentation à une autre ; avec certains Pascal, il existe une confusion entre les termes qui sont redéfinissables par l'utilisateur et ceux qui ne le sont pas. A ce sujet, le Turbo est très clair ; ces termes sont appelés *identificateurs standard.*

Identificateurs standard

Les identificateurs standard sont des mots dont la fonction est prédéfinie en Turbo Pascal ; à la différence des mots réservés, ils peuvent être redéfinis par l'utilisateur.

Mots réservés classés par ordre alphabétique :

ABSOLUTE	AND	ARRAY	BEGIN	CASE
CONST	DIV	DO	DOWNTO	ELSE
END	EXTERNAL	FILE	FORWARD	FOR
FUNCTION	GOTO	INLINE	IF	IN
LABEL	MOD	NIL	NOT	OVERLAY
OF	OR	PACKED	PROCEDURE	PROGRAM
RECORD	REPEAT	SET	SHL	SHR
STRING	THEN	TYPE	TO	UNTIL
VAR	WHILE	WITH	XOR	

Il est plus intéressant de grouper ces mots réservés suivant leurs fonctions :

Définition de données et de structures :

ARRAY	CONST	FILE	NIL	RECORD
SET	STRING	TYPE	VAR	

Structure de programme et contrôle :

BEGIN	CASE	DO	DOWNTO	ELSE
END	EXTERNAL	FORWARD	FOR	FUNCTION
GOTO	INLINE	IF	LABEL	OVERLAY
OF	PROCEDURE	PROGRAM	REPEAT	THEN
TO	UNTIL	WHILE	WITH	

Opérateurs logiques et arithmétiques :

AND	DIV	IN	MOD	NOT
OR	SHL	SHR	XOR	

Adressage mémoire :

ABSOLUTE

Figure 2.1 : Mots réservés du Turbo Pascal.

La possibilité de redéfinir les identificateurs standard est encore une caractéristique du Turbo permettant de résoudre des problèmes de la manière la plus naturelle possible ; il n'est pas nécessaire de reprendre les données pour aboutir à la solution avec une certaine maladresse et d'une manière mécanique dictée par les restrictions matérielles de l'ordinateur et le manque de souplesse du langage du compilateur. L'utilisateur doit obtenir la résolution de son problème, après une approche normale, sans se soucier de l'ordinateur.

Par exemple, la syntaxe standard de la fonction SIN renvoie le sinus d'un angle exprimé en radians. Dans un programme de navigation céleste, l'utilisateur est habitué à exprimer la valeur des angles en degrés plutôt qu'en radians ; il devient donc naturel pour le programmeur de redéfinir la fonction SIN pour qu'elle puisse manipuler des degrés. La redéfinition de la nature de la fonction est beaucoup plus simple que la modification des données avec un sous-programme de conversion de degrés en radians.

Il faut prendre conscience du fait que ce genre de modification éloigne plus ou moins les programmes du Turbo standard. Mais pour les programmeurs qui ont souvent été confrontés à d'autres langages dont les commandes ne leur permettaient pas de faire exactement ce qu'ils voulaient, cette possibilité est très appréciable.

Les identificateurs standard couvrent une grande variété de sujets (constantes, variables, types de données, procédures, fonctions) et de nombreuses possibilités de contrôle de la machine. Toutes les commandes d'entrées-sorties, de traitement de données sur l'écran, de manipulation de fichiers et de restriction sur la machine sont des identificateurs standard.

Identificateurs

L'immense liberté laissée aux programmeurs pour le choix des identificateurs leur apporte une joie nouvelle. Dans le premier exemple, le programme s'appelait Affichage_de_salutations et affichait un message comme l'aurait fait un programme BASIC qui aurait été appelé SALUT.BAS.

Le Turbo accepte une variable dont le nom est, par exemple : CardinalEquipeSaisonBatailleMoyenne ; les identificateurs peuvent contenir jusqu'à 127 caractères et tous les caractères (à l'inverse de ceux de certains BASIC) sont significatifs. Tous les autres Pascal autorisent des identificateurs dont la taille varie suivant l'implémentation, mais seuls les 8, 16 ou 32 premiers caractères sont lus. Avec le Turbo, les identificateurs :

Novembre_Profit

et :

Novembre_Perte

peuvent être utilisés pour représenter deux variables différentes, alors que le Pascal standard ne prend en compte que les huit premiers caractères :

Novembre

et considère que les deux noms se réfèrent à la même adresse mémoire. On conseille souvent aux programmeurs de ne pas exploiter cette possibilité (usage d'indentificateurs longs) pour conserver la portabilité de leurs programmes. C'est un peu comme si on demandait aux programmeurs de ne plus faire de calculs sur des ordinateurs sous prétexte que certains individus emploient des bouliers ou des règles à calcul. L'utilisation d'identificateurs représentatifs est le moyen permettant de rendre les programmes clairs et compréhensibles.
Il faut se rappeler qu'un nom de variable comme Nombre DeTrésorsDansLaCave, NombTrés ou simplement N sera toujours converti par le Turbo en une simple adresse mémoire. La représentation de l'identificateur par un nom est uniquement destinée à rendre le programme plus lisible pour le programmeur ou pour un utilisateur quelconque.
Il faut appliquer la même logique pour nommer des procédures ou des fonctions. La nature de la tâche à exécuter est plus évidente dans un programme appelant une procédure nommée CommandeInterprétation ou Affichage_de_salutations plutôt qu'une liste de lignes de codes ou un numéro de ligne (commande GOSUB).

Règles pour la création d'identificateurs

Plusieurs règles régissent la création d'identificateurs. Nous savons déjà que leur taille ne doit pas dépasser 127 caractères. Bien qu'ils puissent être constitués de nombres, ils ne peuvent commencer que par une lettre ou par le signe souligné (_) ; de plus, ils ne doivent pas contenir d'espace. On remarque que l'emploi de majuscules et de minuscules dans un nom de programme composé de mots agglutinés contribue à la lisibilité et à la compréhension (nous avons vu précédemment que le Turbo ne fait pas de distinction entre les majuscules et les minuscules). Certains programmeurs préfèrent utiliser le caractère souligné comme séparateur de mots ; pour le Turbo, il ne représente qu'un caractère quelconque. Les identificateurs :

CardinalEquipeSaisonBatailleMoyenne

et :

Boxe_Française

sont autorisés. Le choix est laissé à l'appréciation du programmeur.

Les utilisateurs qui ne connaissent que le BASIC pensent parfois que seules les variables ont un nom ; le Turbo permet de nommer de nombreux éléments d'un programme : variables, labels, constantes, types de données, procédures et fonctions. Ainsi un programme Turbo contient non seulement des noms destinés à faciliter les manipulations de données, mais aussi des noms relatifs aux structures de données et aux modules du programme. Ces noms contribuent à la grande clarté des programmes Turbo ; il suffit par exemple d'éditer et de consulter le programme CALC.PAS qui est fourni avec la disquette Turbo Pascal.

Les symboles

Les langages naturels utilisent des jeux de caractères différents ; par exemple, les Allemands n'utilisent pas le caractère tilde (˜), les Espagnols n'emploient pas le tréma (¨) et les Anglais ne connaissent aucun des deux. Le Turbo quant à lui possède son propre jeu de caractères, qui est détaillé dans la Figure 2.2.

Comme en BASIC, le signe > signifie "supérieur à ", tandis que

“différent de ” est représenté par <>. Le Turbo comporte des combinaisons de caractères qui lui sont propres :

:=

et :

(* *)

Ce sont respectivement les symboles d’assignation et de commentaire. Il existe aussi des mots appelés *opérateurs* et qui sont considérés comme des symboles (DIV, MOD, AND, SHL, SHR, OR, XOR, IN) ; ils seront étudiés ultérieurement.

Le Turbo, à la différence d’un langage assembleur, ne réserve pas certains signes à des représentations ou à des fonctions par-

```
Bien que l'on puisse utiliser n'importe quelle combinaison de bits dans les
données d'un programme en Turbo (sur des ordinateurs comme l'IBM PC, il est
possible d'employer des valeurs binaires ou bien les caractères graphiques
complémentaires), seulement certains caractères sont reconnus par le
compilateur. Les instructions d'un programme ne doivent pas comporter de
caractères différents des suivants :

  Toutes les lettres de l'alphabet :

  ABCDEFGHIJKLMNOPQRSTUVWXYZ

  Tous les chiffres décimaux :

  0123456789

  Les symboles suivants :

  .,:.'#$*/+-=<>^[](){}

  Ces symboles ont les significations suivantes :
```

Figure 2.2 : Jeu de caractères du Turbo Pascal.

Opérations mathématiques :

*	multiplication
/	division
+	addition
-	soustraction ou signe moins
()	modification de l'ordre des opérations

Construction de programme :

:=	assignation
.	délimiteur de fin de programme
;	délimiteur de fin d'instruction
#	les caractères suivants sont représentés en ASCII
$	les caractères suivants sont représentés par leur code en hexadécimal
(* *)	commentaire ou directive de compilation
{}	commentaire ou directive de compilation
+	activation des options de compilation
-	désactivation des options de compilation
'	début et fin d'un caractère ou d'une chaîne de caractères

Comparaisons et tests :

=	égalité
<	inférieur
>	supérieur
<=	inférieur ou égal
>=	supérieur ou égal
<>	différent

Structure de données :

^	pointeur ; utilisé aussi pour les codes de contrôle du clavier
[]	indexation ou dimension d'un tableau ; énumération de jeux

Figure 2.2 (suite)

(. .)	indexation ou dimension d'un tableau ; énumération de jeux
()	liste de paramètres pour des fonctions ou des procédures
.	délimiteur de champs dans un enregistrement ; séparateur du nom d'un fichier et de son extension
..	rang

Le Turbo utilise aussi certains mots comme des symboles.

Opérations logiques et manipulations de bits :

NOT	NON logique ou inversion
AND	ET logique
OR	OU logique
XOR	OU exclusif
SHL	déplacement à gauche (multiplication)
SHR	déplacement à droite (division)
IN	inclusion dans un jeu

Opération mathématiques sur les entiers :

DIV	division entière ; donne le quotient
MOD	division entière ; donne le reste

Figure 2.2 (suite)

ticulières ; par exemple, les signes "pour cent " (%), "a commercial " (@), "et commercial " (&), etc. N'ayant pas plus de signification qu'un caractère quelconque, ils peuvent être utilisés sans problème dans les données d'un programme.

Par exemple, le Turbo n'accepte pas d'identificateur portant le nom :

Ventes %Gain

ou bien :

AñoPróximo

tandis que l'instruction suivante est autorisée :

WRITELN("Ventes %Gain")

Le compilateur Turbo a été conçu pour interpréter et donner une signification particulière à un groupe limité de symboles. Cela ne veut pas dire qu'un programme Turbo ne peut pas utiliser comme données un jeu de caractères plus important.

Cette distinction importante montre encore la différence qui est faite par le Pascal entre les données et les instructions de programme. Le Turbo accepte n'importe quel code ASCII pour des données. Le programmeur averti pourra étendre ce concept pour manipuler n'importe quelle combinaison de bits. Rien ne l'empêchera d'écrire un traitement de texte français ou un programme de communication permettant de transmettre des informations purement binaires. En fait, lorsque l'on a indiqué au Turbo le rang des caractères à utiliser, les données peuvent être classées aussi bien par ordre alphabétique que suivant l'ordre spécifié. Ce sujet sera développé dans le chapitre concernant les types de données.

En résumé, le jeu de caractères spécifique des instructions du Turbo est un sous-ensemble du jeu de caractères utilisables comme données dans un programme.

Codes de contrôle et nombres non décimaux

Le Turbo permet d'utiliser comme données des caractères ne faisant pas partie du jeu de caractères alphanumériques. Les séquences de contrôle (Ctrl-Z pour fermer un fichier, Ctrl-G pour émettre un bip, Ctrl-J/Ctrl-M qui représentent le saut de ligne et le retour chariot,...) peuvent être entrées à partir du clavier en utilisant la touche "accent circonflexe " (^) pour matérialiser la touche Ctrl. Par exemple, Ctrl-J est représenté par ^J et Ctrl-Z devient ^Z.

Pour obtenir une séquence Ctrl-G (émission d'un bip), on tape un accent circonflexe (^) suivi du caractère G. L'instruction suivante génère un bip lors de l'exécution du programme :

WRITELN (^G) ;

N'importe quel caractère ASCII peut être représenté par son code numérique ; il suffit de le faire précéder du signe dièse : #.

Par exemple, #65 représente la lettre A majuscule, #126 correspond au signe tilde (˜) et #7 représente Ctrl-G (qui génère un bip).

Comme le signe dièse indique au Turbo que le nombre qui suit représente un code ASCII, le signe dollar ($) précède une valeur en hexadécimal ; par exemple, le nombre hexadécimal 3F (63 en décimal) devient $3F.

Longueur des lignes et délimiteurs

En Turbo, il est impossible de prolonger une ligne (*bouclage*) ; une simple ligne de code ne peut contenir plus de 127 caractères. C'est plus un avantage qu'une restriction. Le Turbo offre toutes les possibilités de construction envisageables pour manipuler de longs messages de texte ou pour résoudre des problèmes complexes sans utiliser des lignes de code qui bouclent de manière confuse sur l'écran. Ainsi, en dépit des habitudes des programmeurs, les programmes sont rendus plus lisibles.

D'autre part, les éléments individuels du langage doivent être séparés par un espace (ce qui explique que ceux-ci soient interdits à l'intérieur d'un identificateur). Le retour chariot à la fin d'une ligne sert aussi de séparateur, comme l'insertion d'un commentaire.

Les programmeurs BASIC "durs " qui sont habitués à tasser les lignes (pour ne pas occuper trop de place en mémoire) peuvent abandonner définitivement ces méthodes. Il faut se rappeler que la compilation transforme un programme source lisible en un programme compact en code machine (ce type de fichier porte l'extension .COM et occupe une zone mémoire beaucoup plus réduite).

La possibilité d'indentation automatique offerte par l'éditeur du Turbo entraîne l'insertion d'une grande quantité d'espace dans le programme source.

Commentaires

L'utilisation d'identificateurs dont les noms sont imagés contribue à rendre les programmes plus compréhensibles ; mais, comme pour tous les langages de programmation, l'emploi de commentaires s'avère indispensable. Ceux-ci peuvent être placés à n'importe quel endroit du programme. Leur taille et leur fréquence

n'influent en rien sur le déroulement du programme ; ils ne sont pas pris en compte lors de la création du code exécutable.

En Turbo, les commentaires sont placés entre accolades :

{**Voici un commentaire en Turbo**}

ou bien entre parenthèses et astérisques, comme en Pascal Standard :

(*Voici encore un commentaire en Turbo*)

Il est impossible d'imbriquer des commentaires dans des commentaires à l'aide d'accolades si les délimiteurs externes sont aussi des accolades ; il en est de même avec des parenthèses et des astérisques si les délimiteurs externes sont aussi des parenthèses et des astérisques. Cependant, l'imbrication peut être réalisée si les délimiteurs internes sont différents des délimiteurs externes. Cela est très intéressant lors de la mise au point de programmes. Des zones entières de code peuvent être ainsi séparées du reste du programme (la Figure 2.3 illustre cette possibilité ; les commentaires sont placés entre accolades et une zone de code et de commentaires est placée entre parenthèses et astérisques).

Lors de la mise au point du programme de la Figure 2.3, on peut éliminer une des options (la possibilité de faire un tri par ordre alphabétique ou suivant le code postal). On remarque l'utilisation de parenthèses et d'astérisques (* et *) ; si l'on avait utilisé des accolades ({ et }), le compilateur aurait été dérouté par le commentaire :

{**test seulement si "A "**}

En effet, les délimiteurs de commentaires externes auraient été identiques aux délimiteurs de commentaires internes. En isolant de cette manière des blocs entiers de code puis en supprimant les délimiteurs externes au fur et à mesure de la mise au point, il est possible de réaliser des programmes très performants.

Le Turbo utilise aussi les délimiteurs de commentaires pour d'autres tâches : les directives de compilation. Elles doivent être placées en début de ligne ; leur syntaxe est étudiée dans le chapitre réservé aux options de compilation.

```
PROCEDURE Entree_a_partir_de_fichiers_externes;

BEGIN
   CLRSCR; {efface l'écran}
   WRITELN ('Nom du fichier contenant les données à trier ? ');
   READLN (Fichier_source);
   ASSIGN (Fichentree, Fichier_source);
   WRITELN ('Nom du fichier qui contiendra les données triées ? );
   READLN (Fichier_dest);
   ASSIGN (Fichsortie, Fichier_dest);
   RESET (Fichentree);
   Nuenr := 0;
   Alpha_seul := FALSE; {Si l'utilisateur ne veut pas de tri alphabétique}
   (*
   WRITELN ('Taper A pour un tri alphabétique ; Z pour un tri par code postal');
   READLN (Alpha_ou_code);
   IF Alpha_ou_code = 'A' THEN Alpha_seul := TRUE; {test seulement si "A"}
   *)
   WHILE NOT EOF(Fichentree) DO {chargement d'un tableau avec les noms}
   BEGIN
      READ (Fichentree,Client);
      Nuenr := Nuenr + 1;
      Num := Client.code;
      IF Alpha_seul THEN Num := '0';
      Tri[Nuenr].Code_et_nom := Num + Client.Nom;
      Tri[Nuenr].Cle := Nuenr;
   END;
   Maxenr := Nuenr;
   CLOSE (Fichentree);
END;
```

Figure 2.3 : Utilisation de différents délimiteurs de commentaires.

FORMAT DES PROGRAMMES EN TURBO

Le Turbo Pascal est plus qu'un jeu de conventions de programmation ; c'est la concrétisation d'une philosophie qui s'appuie sur la clarté, la simplicité et l'intelligibilité. Les règles syntaxiques qui ont déjà été abordées nous seront utiles pour appréhender la manière dont le compilateur travaille et connaître ses limites.

Le Turbo, à la différence de certains langages de programmation, possède de nombreuses conventions complexes, non pas à cause des diverses possibilités de manipulation et de résolution proposées par le compilateur, mais parce que la structure requise pour les commandes est très proche du langage naturel.

Le lecteur ayant ces notions présentes à l'esprit verra dans un programme en Turbo Pascal le cheminement conduisant à la résolution d'un problème sans être irrité par les défauts du compilateur. Le Turbo demande simplement que les éléments (type des données, variables, constantes et procédures) soient déclarés avant que le jeu commence.

Cela n'est pas plus gênant que les règles du jeu que doivent connaître et appliquer les arbitres d'un sport quelconque au cours d'une rencontre. Cela n'est pas plus artificiel que la liste de courses que nous rédigeons avant d'aller au supermarché ou chez l'épicier du coin, ou les vêtements que nous préparons le soir en prévision de la réception à laquelle nous devons nous rendre le lendemain. Comme ces procédures qui sont intégrées à la vie de tous les jours, les règles à appliquer pour l'écriture d'un programme en Turbo permettent de réaliser de nombreuses tâches.

Avant d'aborder le format d'un programme en Turbo, imaginons la démarche à suivre pour construire un abri de jardin. Tout d'abord, il faut se représenter ce que l'on veut construire et généralement en réaliser un dessin (un algorithme). Ensuite, on établit une liste des matériaux à acheter pour commencer le chantier. Le problème des outils se pose : ceux que l'on a déjà, ceux que l'on doit acheter et ceux que l'on doit louer. Enfin, lorsque nous sommes en possession des plans, des matériaux et des outils, il ne reste plus qu'à commencer la construction du bâtiment.

En Turbo, on suit un processus analogue. La liste de matériel devient une partie des déclarations du programme (liste des données qui seront utilisées). Le rassemblement des outils est semblable à la création de procédures spécifiques destinées à manipuler les données. A la fin, le corps du programme utilise les outils (procédures) avec le matériel (données) pour construire la maisonnette (résolution du problème).

La Figure 2.4 illustre l'organisation d'un programme en Turbo. Évidemment, tous les programmes n'ont pas une structure identique à celle-ci, mais les plus complexes sont composés de parties identiques. Il est intéressant de jeter un coup d'œil sur le programme CALC.PAS pour voir comment un programme relativement complexe se conforme à cette organisation.

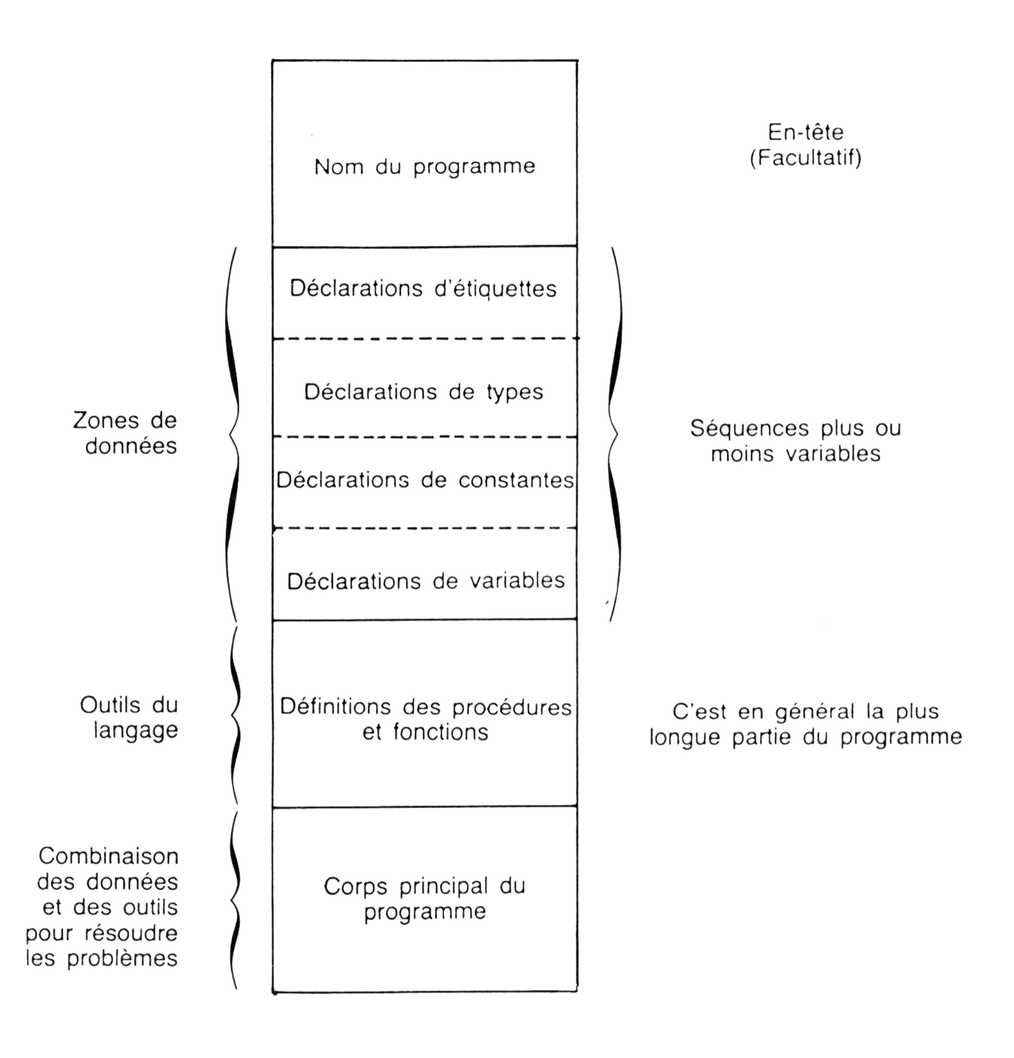


Figure 2.4 : Organisation d'un programme en Turbo.

La Figure 2.5 est le listing d'un programme simple qui calcule le nombre de billets de bus nécessaires à un groupe de supporters au cours d'une saison de football (connaissant le nombre de rencontres qui ont lieu). Naturellement, il n'est pas indispensable d'écrire un tel programme pour faire une simple multiplication ; cependant, ce programme (appelé Tickets_de_bus) illustre plusieurs points importants de la programmation en Turbo.

```
PROGRAMM Tickets_de_bus;

VAR
    membres, rencontres, tickets : INTEGER;

BEGIN
    WRITELN ('Nombre de membres dans l amicale ? ');
    READLN (membres);
    WRITELN ('Nombre de rencontres ? ');
    READLN (rencontres);
    Tickets := membres * rencontres;
    WRITELN;
    WRITELN ('Il faudra ', Tickets, ' tickets de bus pour cette saison.');
END.
```

Figure 2.5 : Exemple de programme en Turbo.

En-tête de programme

L'en-tête de programme contient le nom du programme et éventuellement des informations relatives aux fichiers utilisés en entrée ou en sortie. En Turbo, l'en-tête est facultatif tandis qu'en Pascal Standard, il est obligatoire. L'utilisation d'un en-tête permet de donner au programme un nom très explicite afin que la seule lecture de l'en-tête informe l'utilisateur de la tâche qu'il accomplit. Il faut se rappeler que le nom d'un fichier DOS est limité à huit caractères et que son extension ne doit pas dépasser trois caractères. Voici des exemples de noms :

Noms de fichiers DOS	**Noms de programmes Turbo**
EXEMPLE.PAS	Exemple_de_calculs_statistiques
COMPLEXE.PAS	Calculs_sur_les_nombres_complexes
EXEMPLE2.PAS	Interets_sur_locations_de_voiture

Peu importe la taille de l'en-tête, il n'est pas pris en compte par le compilateur. Le programme de la Figure 2.5 a pour en-tête : Tickets_de_bus.

Zone de déclaration

Le Turbo impose que chaque élément de données et chaque module du programme soit déclaré avant d'être utilisé lors de l'exécution du programme. Les déclarations de données servent simplement à indiquer au Turbo quelles données il doit accepter au cours de l'exécution du programme. Il est nécessaire que le compilateur puisse réserver en mémoire un espace suffisant pour réaliser d'une manière efficace les tâches qui lui sont confiées. Comme certains types de données nécessitent plus de mémoire que d'autres (il est évident que le nombre Pi, stocké sur huit chiffres : 3,14159265, occupera plus de place que le nombre entier 3 ou le code du caractère Q), il faut indiquer au compilateur non pas la valeur de la donnée mais son type. Ces informations sont indispensables pour la réservation de l'espace mémoire et pour contrôler le chemin que le compilateur doit suivre pour exécuter différentes opérations mathématiques. La connaissance et la manipulation des différents types de données étant relativement délicates en Turbo, elles seront développées ultérieurement dans un chapitre particulier.

La syntaxe de la déclaration de type de données est très simple ; les identificateurs sont séparés du type de données par deux points. De nombreux exemples sont donnés au cours de cet ouvrage.

Le rapport entre le nom des variables et leur type est aussi une chose très importante. Dans le programme de la Figure 2.5, les variables s'appellent : Membres, Rencontres et Tickets. L'information qui sera représentée par ces variables est manifeste ; chacune de ces variables correspond à un nombre. Dans un autre programme, l'identificateur Membres pourra se référer à une chaîne de caractères représentant une liste de noms plutôt qu'un nombre.

Dans les déclarations de cet exemple, on indique au compilateur que les nombres associés à chaque variable sont des nombres entiers. Ainsi, si on tente de donner à la variable Membres la valeur 7,3 (qui n'est pas un nombre entier), le compilateur Turbo indiquera immédiatement une erreur. De la même manière, si l'on écrit la liste des membres de l'équipe à la place d'un nombre, cette valeur sera refusée. Le compilateur teste immédiatement la *validité* de l'entrée. Certains langages évolués mettent ainsi l'accent sur la

nature des données en développant le concept de type de données.

Le concept de type de données

La nécessité de déclarer le type de données est sans doute l'aspect du Turbo Pascal qui a donné lieu aux discussions les plus mouvementées. De nombreux programmeurs habitués aux libertés offertes par le BASIC considèrent cette notion comme une nuisance et en concluent que le Pascal est un langage embarrassant et ennuyeux.

La déclaration de type est un concept précieux qui conduit à une clarification de la pensée ; elle caractérise de nombreux langages modernes (Pascal, C, Modula2 et ADA). Personne ne peut prétendre qu'un programmeur dont les pensées sont claires et structurées concevra inévitablement des programmes dont la mise au point et les modifications ultérieures seront délicates. De plus, le fait de déclarer le nom et le type des variables simplifie grandement le travail du compilateur. L'efficacité, la simplicité et la variété des compilateurs ont fait du Turbo un produit très intéressant et très puissant. On comprend aisément pourquoi tous ces langages structurés sont de plus en plus populaires pour le développement de logiciels commerciaux. Ils apportent une plus grande performance avec une complexité moindre.

A tous les niveaux, les déclarations de types de données évitent les problèmes. Certaines des erreurs les plus difficiles à déceler dans des programmes BASIC proviennent simplement d'une non-concordance de type ou de fautes d'orthographe. Parfois, alors que l'on pense manipuler la même variable dans un programme, une faute d'orthographe s'est glissée dans son nom et si le langage utilisé est le BASIC, elle est traitée comme une seconde variable. Si le programme est écrit en Turbo, le compilateur détecte immédiatement l'erreur et indique à l'utilisateur que l'identificateur n'a pas été déclaré (il ne le trouve pas sur la liste des variables).

Des erreurs plus insidieuses apparaissent parfois dans de longs programmes et sont très difficiles à dépister. Le programmeur peut oublier qu'il a utilisé au début d'un programme le nom de variable LIMITE ou DEBUT et l'employer à nouveau. A moins que l'on prenne la précaution d'initialiser les variables chaque fois qu'elles sont utilisées, il est difficile de prévoir le résultat d'un programme si un même nom est employé pour deux variables différentes. Le Turbo empêche ce genre d'erreur en indiquant à l'utilisateur qu'il y a une définition double.

La déclaration de types de variables n'est pas une notion incon-

nue en BASIC ; si la variable est de type alphanumérique, elle doit se terminer par le caractère dollar ($), les chaînes de caractères doivent être placées entre guillemets et, suivant les types de précision requis par une variable numérique, son nom doit se terminer par un caractère particulier (%, ! ou #). Ces caractères spéciaux ont le même rôle que les déclarations de types en Turbo. Dans de longs programmes qui occupent une place mémoire importante, les programmeurs utilisent des variables de type entier pour définir les variables des boucles de comptage (par exemple : FOR J %=1 TO 17) ; par défaut, les variables sont de type réel simple précision et occupent une place mémoire plus importante. Dans les programmes plus ambitieux, on remarque souvent l'instruction DEFINT qui sert à déclarer des variables de type entier.

Le Turbo encourage le programmeur à faire le meilleur usage de la mémoire de l'ordinateur ; il doit déclarer le type des variables qu'il va utiliser. Comme nous le verrons plus tard, il faut penser à la précision des calculs que le Turbo doit effectuer (en particulier pour les programmes comptables relatifs à des sommes d'argent et dont la précision est le centime). Un programme en Turbo indique immédiatement une entrée invalide et n'entreprend pas de calculs si les valeurs n'ont pas de sens.

Après avoir écrit quelques programmes en Turbo, la déclaration des variables n'est qu'une simple formalité et devient un réflexe. Chaque fois que l'on commence à travailler avec des structures de données, on s'étonne qu'il y ait encore des individus qui se servent de langages de programmation ne se souciant pas des types de données.

Constantes

En plus de l'identification des variables, le Turbo nécessite celle de tous les autres éléments de données. Les plus communs sont les *constantes*. Les constantes sont simplement des valeurs spécifiques associées à des identificateurs créés par l'utilisateur ; elles peuvent être de n'importe quel type. Voici quelques exemples :

```
Pi = 3.1416                      (* Nombre reel *)
Tresor = 25                      (* Entier *)
Taux_taxe = 0.05                 (* Nombrereel *)
Bip = #07                        (* Code caractere *)
Message = 'Je ne connais pas ce mot !' (* Chaine *)
```

Les constantes ont deux fonctions en Turbo. La plus commune

est de représenter une valeur constante et qui ne change donc pas au cours de l'exécution du programme. La seconde est une aide au programmeur pour initialiser des variables ou bien pour leur assigner une valeur initiale. Cela permet de connaître immédiatement la valeur de départ et d'éviter de charger le programme avec des instructions d'initialisation placées plus loin.

Les valeurs des constantes de l'exemple précédent ne peuvent pas être changées dans le programme. Si on veut utiliser les définitions de constantes pour initialiser les valeurs, il suffit de modifier les déclarations de manière à inclure les types de données et les valeurs initiales :

```
Pi : REAL = 3.1416 ;
Trésor : INTEGER = 25 ;
Taux_taxe : REAL = 0.05 ;
Bip : CHAR = #07 ;
Message : STRING [80] = 'Je ne connais pas ce mot !' ;
```

Déclaration de structures

Nous avons vu que le Turbo impose la déclaration d'un identificateur et de son type pour chaque élément de données. Il faut aussi déclarer les procédures, fonctions et périphériques qui doivent être utilisés, c'est-à-dire tous les modules de codes qui permettront au programme de résoudre le problème. Le programmeur qui emploie le Turbo et ses outils (procédures et fonctions) ne doit rien déclarer ; les fonctions du Turbo standard incluent les commandes de base de l'écran et d'entrée/sortie sur disque, ainsi que les fonctions trigonométriques et une multitude d'autres utilitaires.

Néanmoins, même le plus élémentaire des programmes contient quelques définitions de modules. Ces modules (procédures et fonctions) peuvent être regroupés et constituer ultérieurement les archives qui serviront à l'élaboration de nombreux programmes.

Il est temps maintenant d'étudier un programme un peu plus complexe que les précédents ; le programme de la Figure 2.6 appelé Titre_courant est constitué de structures qui n'ont pas encore été abordées.

Procédures et fonctions

Dans un programme Turbo typique, l'essentiel du code est

contenu dans des modules appelés *procédures*. Nous avons vu que les *fonctions* sont étroitement associées au concept général de procédure. Une fonction, par définition, produit une valeur qui peut être substituée au contenu d'une variable. Une procédure peut aussi générer une valeur, mais elle fait bien d'autres choses ; il n'est pas nécessaire qu'elle substitue une valeur à celle d'une variable. L'essentiel de cet ouvrage est consacré à l'étude des procédures et des fonctions.

PROGRAM Titre_courant ; { en-tête }

VAR { déclaration }

```
PROGRAM Titre_courant;                    {En-tête}

VAR                                       {Déclarations}
   Nombre_Etoiles : INTEGER;
   Message : STRING[76];
   Point_depart : INTEGER;

PROCEDURE Ligne_Etoiles (Largeur : INTEGER);
VAR
   Compteur : INTEGER;
BEGIN
   FOR Compteur := 1 TO Largeur DO WRITE ('*');
   WRITELN;
END;

PROCEDURE Message_utilisateur;
BEGIN
   WRITELN ('Tapez votre message ?');
   READLN (Message);
END;

BEGIN                                     {L'exécution du programme commence ici}
   CLRSCR;
   Message_utilisateur;
   CLRSCR;
   Nombre_Etoiles := LENGTH(Message) + 12;
   Ligne_Etoiles (Nombre_Etoiles);
   WRITELN ('*     ',Message,'     * ');
   Ligne_Etoiles (Nombre_Etoiles);
END.
```

Figure 2.6 : Programme Turbo utilisant de nombreuses procédures.

Étiquettes

Le Turbo permet aussi la déclaration d'*étiquettes* ; elles sont destinées à repérer des emplacements dans un programme afin de pouvoir employer l'instruction GOTO familière aux utilisateurs du BASIC. Les étiquettes doivent être déclarées au début du programme avec le mot réservé LABEL, et groupées dans la zone des procédures et des fonctions. Elles sont utilisées par le système pour contrôler le déroulement du programme. Les étiquettes sont rarement employées en Turbo parce que l'instruction GOTO n'apparaît pratiquement jamais dans un programme structuré. En fait, la plupart des programmeurs Turbo évitent l'instruction GOTO non seulement pour des problèmes de clarté mais aussi parce qu'elle est incompatible avec la notion de programmation structurée. Le programmeur qui tient à se servir de l'instruction GOTO doit penser à déclarer les étiquettes. A la différence d'autres langages Pascal, le Turbo applique aux étiquettes les mêmes règles qu'aux identificateurs créés par l'utilisateur. De plus, les étiquettes peuvent commencer par un chiffre et leur taille n'est pas limitée (en Pascal standard, elles sont limitées à quatre chiffres).

Le corps principal du programme

Le corps principal d'un programme en Pascal tend en général à être la partie la plus courte. Il est constitué de commandes de début et de fin de gestion de tâches (ouverture et fermeture de fichiers, et parfois affichage de messages relatifs aux sous-programmes) et d'appels de procédures qui réalisent tout le travail du programme. Le corps principal du programme débute avec le mot réservé BEGIN et se termine par le mot réservé END (suivi d'un point). Le nom des procédures auxquelles il fait appel renseigne le lecteur sur la nature des tâches réalisées par le programme (voir le programme de la Figure 2.7).

Résumé sur les programmes structurés

Les programmes commencent par un en-tête facultatif qui identifie le programme. Viennent ensuite les déclarations d'étiquettes et de données (variables, constantes et structures de données), puis les déclarations relatives au déroulement du programme : procédures et fonctions. A la différence d'autres Pascal, le Turbo est très souple quant au format des déclarations ; les variables,

```
BEGIN                                   {début de l'exécution du programme}
   Tout_fini := FALSE;
   Initialise;
   WHILE NOT Tout_fini DO
      BEGIN
         Affiche_menu;
         Interprete_choix;
      END;
END.
```

Figure 2.7 : Corps principal d'un programme en Turbo.

constantes et étiquettes peuvent être déclarées dans n'importe quel ordre. Ainsi, dans les programmes longs et complexes, le programmeur a la possibilité de fragmenter ses déclarations et de les répartir à plusieurs endroits du programme. Si les déclarations de données sont structurées en fonction des procédures auxquelles elles se rapportent, le programme est beaucoup plus facile à comprendre.

Enfin, la dernière partie d'un programme en Turbo est le corps principal du programme ; elle est essentiellement constituée d'appels aux procédures définies auparavant.

Ponctuation

Nous avons déjà vu que le Turbo requiert certaines règles précises concernant la ponctuation. Ces règles seront mieux appréhendées au fur et à mesure de l'apprentissage du langage ; cependant, les règles majeures sont résumées ci-dessous :

- Une déclaration de *constante* utilise le signe égal et se termine par un point-virgule :

  PI = 3.1416 ;

- Une déclaration de *variable* utilise deux-points et se termine par un point-virgule :

  Chambres : INTEGER ;

- Lorsque la déclaration est constituée d'une liste, les éléments sont séparés par une virgule :

Ogres, Voitures, Chambres_vides : INTEGER ;

Prix_de_vente, Taux_taxe : REAL ;

On peut s'étonner de trouver des lignes qui se terminent ou non par un point-virgule. En Turbo, le point-virgule est un *séparateur d'instructions* ; dans la plupart des cas, cela signifie qu'il apparaît à la fin de chaque ligne de code. Dans certaines structures de contrôle (IF / THEN / ELSE qui sont les plus communes), les options individuelles ne sont pas séparées par des points-virgules ; il en est de même pour la dernière instruction END du programme (elle doit être suivie d'un point).

Les programmeurs débutants ont en général un peu de mal à assimiler tout de suite ces règles de programmation. Cela n'est pas très important dans la mesure où le compilateur Turbo placera des messages d'erreur là où la ponctuation est incorrecte. La recherche et la correction d'erreurs sont un plaisir avec le Turbo alors qu'avec d'autres Pascal elle deviennent des tâches longues et difficiles. Pour l'utilisateur, le meilleur moyen de se familiariser avec la ponctuation consiste à étudier les exemples de programmes puis à étudier les messages d'aide proposés par le compilateur lors de l'écriture de ses propres programmes. La ponctuation correcte devient rapidement une seconde nature.

COMPLÉMENT SUR LES VARIABLES

Les variables ont été rapidement abordées au début de ce chapitre, mais sans faire la distinction entre les variables algébriques et les variables des langages informatiques. Le Turbo utilise le terme *variable* pour se référer à un emplacement de la mémoire de l'ordinateur dans lequel est placée une valeur correspondant à un élément de données. En général, cette valeur est inconnue durant tout ou partie de l'exécution du programme ou bien elle est modifiée au cours de son déroulement. Les programmes sont écrits pour trouver le résultat de manipulations ou de calculs de quelques variables.

Le Turbo est très souple dans la mesure où il est possible d'identifier les données qui sont utilisées dans un programme. Cette flexibilité s'applique aux variables, aux constantes et à la totalité des structures de données. Nous avons déjà insisté sur la nécessité

d'indiquer au Turbo le type de données associé aux identificateurs. Nous pouvons maintenant explorer les types de données de base du Turbo. Le traitement met l'accent sur le concept de différenciation des types de données et sur leurs applications. Comme pour toutes les règles de syntaxe du Turbo, la meilleure manière d'apprendre le fonctionnement des types de données consiste à écrire des programmes personnels et à laisser le compilateur Turbo placer ses messages d'erreur. La plupart des utilisateurs trouvent fastidieux d'apprendre une série de règles mais la correction de quelques lignes de programme spécifiques aux messages d'erreur permet une assimilation des règles sans effort. Le compilateur Turbo positionne automatiquement le curseur sur la ligne comportant l'erreur et affiche des messages explicites tels que : *Integer constant expected* (attente d'une constante entière), *Invalid result type* (type de résultat non valide) ou *Error in integer constant* (erreur dans la constante entière). Ce type de message rend les applications plus concrètes lorsqu'il s'agit de situations réelles et non hypothétiques.

TYPES DE DONNÉES PRÉDÉFINIS EN TURBO

Les types de données prédéfinis en Turbo Pascal sont *les entiers, les réels, les caractères et les booléens.* Un type de données prédéfini est déjà connu par le compilateur. Il suffit de déclarer une variable comme appartenant à un de ces types pour que le compilateur sache manipuler correctement toute assignation faite ultérieurement par l'utilisateur à cette variable. A l'inverse, lorsque l'on commence à définir des types de données personnels, il faut les spécifier en termes de types de données prédéfinis.

Il ne faut pas oublier que tous les types de données se réfèrent à la fois à des variables et à des constantes.

Entiers

Les *entiers* sont simplement des nombres entiers. Ils peuvent être positifs ou négatifs, compris entre −32768 et +32767. La majorité des boucles et des compteurs sont contrôlés par des valeurs entières :

```
FOR X = 1 TO 10 DO
```

En Turbo Pascal, un entier est stocké en mémoire sur deux

octets. Il ne faut jamais utiliser une virgule à l'intérieur d'un entier, sinon le compilateur Turbo génère un message d'erreur car il "croit " que des chiffres ont été mélangés à des caractères alphabétiques. L'option BCD (*Binary Coded Decimal* : décimal codé en binaire) disponible chez les distributeurs du logiciel offre de grandes possibilités de formatage de résultats numériques. Les utilisateurs qui ne la possèdent pas et qui ont besoin de formater des résultats avec des virgules peuvent écrire leur propre procédure d'analyse d'un nombre pour définir la place des virgules (ou des espaces) puis afficher ce nombre avec les séparateurs en bonne position.

Le Turbo a un identificateur standard, MAXINT, qui est une constante égale à 32767 (l'entier positif le plus grand). Bien sûr, des nombres plus importants peuvent être manipulés, mais si les calculs génèrent un résultat supérieur à 32767, le compilateur Turbo ne signale pas ce "dépassement " ; il exécute le calcul avec des données erronées. Lorsqu'un programme doit utiliser des nombres importants, on peut éviter ce problème de différentes manières, la plus simple consistant à employer à la place des valeurs réelles. Les entiers peuvent être additionnés, soustraits et multipliés à volonté. Dans chaque cas, le résultat des opérations est un entier. Cependant la division peut donner des résultats différents. Par exemple, 8 divisé par 2 a pour résultat l'entier 4, mais 4 divisé par 3 a pour résultat le nombre réel 1.33333. Si l'on veut un résultat de type réel, il n'y a pas de problème ; mais pour obtenir des entiers, Turbo propose deux opérateurs mathématiques spéciaux, DIV et MOD. Ils fonctionnent comme en BASIC. DIV renvoie le quotient entier d'une opération et MOD le reste. Par exemple :

15 DIV 3 = 5
15 DIV 2 = 7 (et non pas 7.5 qui n'est pas un entier)
15 / 3 = 5.0 (nombre réel)
15 / 2 = 7.5 (nombre réel)
15 MOD 3 = 0 (il n'y a pas de reste)
15 MOD 2 = 1

Il est également possible de travailler avec des entiers représentés sous forme hexadécimale. Cette possibilité est particulièrement utile pour l'exploitation des commandes les plus performantes du Turbo qui concernent les manipulations de bits et d'octets ou celles des adresses mémoire (qui sont toujours exprimées en hexadécimal). Les valeurs hexadécimales sont précédées d'un signe $; ainsi le nombre décimal 2634 se présente sous la forme $A4B.

Valeurs d'octets

Le type de données *octet* se réfère à la *sous-catégorie* d'entiers dont les valeurs sont comprises entre 0 et 255. Comme son nom l'indique, une valeur d'octet occupe seulement un octet de mémoire. Les éléments de données d'octets peuvent être mélangés à des entiers. On n'utilise presque jamais ce type de données, excepté pour les programmes dans lesquels il faut conserver un maximum de place mémoire. Dans la plupart des cas, il est plus naturel d'employer le type entier.

Nombres réels

Les *nombres réels* ont deux applications distinctes que ne permettent pas les entiers. La plus importante concerne l'expression de valeurs fractionnaires. Pour exprimer 1.5 ou 0.0000000285, il faut utiliser une représentation de type réel. De plus, les nombres réels peuvent exprimer une gamme de nombres plus vaste (sans astuce de programmation particulière) que les entiers.

Étant donné que les nombres très grands ou très petits sont difficiles à manier (avec de longues chaînes de 0), le Turbo exprime normalement les nombres réels en *notation exponentielle*. Celle-ci comprend une *base* ou *mantisse* et un *exposant*. Quelques exemples de cette représentation apparaissent dans le programme de la Figure 2.8.

Programme	Résultat
BEGIN	
WRITELN (0.12345);	1.2345000000E-01
WRITELN (1.2345);	1.2345000000E+00
WRITELN (12.245);	1.2345000000E+01
WRITELN (0.0000000012345);	1.2345000000E-09
WRITELN (12345000000.0);	1.2345000000E+10
END.	

Figure 2.8 : Programme en Turbo pour représenter une série de nombres réels.

La souplesse et la fiabilité de la notation exponentielle sont plus évidentes lorsque l'on examine le plus grand et le plus petit nombres que le Turbo permet de représenter sous forme de valeurs réelles. Ce sont :

1E−38 et 1E38

Cette représentation est plus pratique que la suivante :

1 * 1/100 000 000 000 000 000 000 000 000 000 000 000 000

et :

1 * 100 000 000 000 000 000 000 000 000 000 000 000 000

Bien que cette méthode soit particulièrement utile pour l'expression de nombres très grands ou très petits, la notation exponentielle est gênante pour l'expression de nombres courants tels que 1.75 F, 1/2 ou même 1.0. Malheureusement le Turbo, comme les autres Pascal, ne propose pas d'alternative pour l'expression de nombres réels de cette forme. Pour une sortie écran ou imprimante, on peut toujours convertir cette représentation en une notation décimale plus familière à l'aide d'une vaste gamme de contrôles de format. Cette conversion est abordée dans la section décrivant les sorties sur écran.

Une valeur réelle est stockée sur 6 octets de mémoire.

Bien que le dépassement de capacité de stockage d'une valeur réelle soit beaucoup moins probable qu'avec une valeur entière, il peut avoir lieu. Si l'on demande à l'ordinateur d'exécuter un calcul qui génère une valeur dépassant ces limites, le programme s'arrête au lieu de continuer avec un résultat non valide. Comme pour les entiers, si l'on utilise des nombres non autorisés, il existe des techniques particulières pour les manipuler ; celles-ci ne sont pas traitées dans cet ouvrage.

Les nombres réels se prêtent à l'exécution de calculs ; toutes les opérations mathématiques de base les emploient. Par contre, les nombres réels ne se prêtent pas à la plupart des structures de contrôle d'un programme. Plus précisément, ils ne peuvent pas être employés pour indexer des tableaux ou pour contrôler des compteurs et des boucles. Il existe quelques autres restrictions relatives à l'utilisation de nombres réels mais elles sont soit évidentes, soit suffisamment obscures pour qu'il ne soit pas nécessaire de s'en occuper en dehors de l'écriture de programmes très sophistiqués.

Les valeurs booléennes

La plupart des programmeurs pensent que les booléens ont uniquement les valeurs VRAI ou FAUX. Cela est partiellement exact ; cependant, le Turbo (comme de nombreux langages de programmation) donne à VRAI la valeur 1 et à FAUX la valeur 0. Cette possibilité d'utiliser soit une valeur numérique, soit les mots VRAI ou FAUX peut conduire à des astuces intéressantes. Les valeurs booléennes ou logiques sont employées exclusivement dans des instructions conditionnelles qui contrôlent le déroulement du programme. Une valeur booléenne occupe seulement 1 octet de mémoire.

Caractères

Nous avons déjà mentionné que les caractères sont des codes ASCII compris entre 0 et 255. Même si l'ordinateur et l'imprimante utilisés ne peuvent pas représenter les caractères dont les codes sont compris entre 128 et 255, il est quand même possible d'employer ces valeurs dans des programmes en Turbo. Le Turbo possède également plusieurs outils puissants pour trier, ordonner et comparer des caractères. D'autre part, il ne faut pas oublier que même si les documents fournis avec le logiciel se réfèrent aux caractères comme des codes ASCII, les programmes peuvent les manipuler sans s'occuper de leur représentation. Par exemple, bien que le Turbo accepte #11 comme code ASCII, ce caractère dirigé vers un périphérique externe peut apparaître comme un caractère EBCDIC ou même comme le caractère *end-of-take* des anciennes télétypes. Un caractère occupe seulement un octet de mémoire.

Remarque sur les types scalaires

Les types de données mentionnés jusqu'à présent (entier, réel, octet, caractère et booléen) sont appelés dans certains ouvrages *types scalaires*. Malheureusement, les différentes sources ne définissent pas ce terme de la même manière et il ne recouvre pas toujours les mêmes éléments. Le terme *scalaire* devrait s'appliquer à tous les types de données composés d'une collection de valeurs *discrètes*. La définition la plus courante comprenant une échelle avec des intervalles fixes implique que toute valeur soit toujours immédiatement précédée et suivie d'une autre valeur. Cette défi-

nition est adaptée aussi bien aux nombres entiers (1, 2, 3, etc.) qu'aux caractères (A, B, C, etc.) et mêmes aux valeurs booléennes (VRAI et FAUX) mais pas aux nombres réels. En effet, quelle valeur précède 3.0 ? Est-ce 2.0, 2.9 ou 2.999999999 ?

D'autre part, du point de vue de la programmation, la définition de scalaires la plus logique est la suivante : les scalaires sont des types de données qui servent à prendre des décisions ou à opérer sur un compteur (choisir l'option A, B ou C ou contrôler la boucle DO classique : FOR X = 1 TO 17 DO). On a donc besoin d'un type de données dans lequel les valeurs ont des pas discrets fixes.

Par ailleurs, certains Pascal (y compris le Turbo) placent les valeurs réelles dans les scalaires et dressent ensuite une liste de restrictions concernant leur utilisation. La raison de cette classification est liée au fait que les nombres réels peuvent servir à contrôler certaines situations (par exemple, "IF P > 2.75 THEN DO quelque chose "). La meilleure attitude consiste à essayer d'ignorer les concepts de scalaires contradictoires mais il ne faut pas oublier, lors de l'établissement de compteurs, de boucles ou de tests, que les nombres réels ne se prêtent généralement pas aussi bien au contrôle du déroulement du programme que les nombres de type scalaire.

Types de données définis par l'utilisateur

Nous reviendrons sur ce sujet lorsque nous aborderons des structures de données plus sophistiquées, mais il faut savoir que les types de données étudiés sont tous des types prédéfinis standard. La plupart des informaticiens classent tous les types de données comme des structures de données en distinguant les structures simples (caractères, entiers, nombres réels, etc.) qui ne peuvent pas être fractionnées en plusieurs unités fondamentales, et les structures de données complexes (chaînes, tableaux, enregistrements, fichiers, etc.) élaborées en combinant des structures simples. Cette représentation est assez analogue à celle des atomes et des molécules. L'utilisateur est parfaitement libre de créer ses propres types et structures de données. Lors de l'écriture d'un programme de formatage de texte, on a parfois besoin d'une structure de donnée appelée *page* composée de plusieurs chaînes. De même, on peut organiser un groupe de valeurs booléennes dans un tableau appelé "Touches_de_réponses " pour corriger des tests vrai/faux.

Si ces exemples semblent un peu artificiels, il est bon de savoir

que l'on peut créer un type de données appelé Semaine constitué d'éléments individuels appelés Lundi, Mardi, Mercredi, etc. Il est également possible d'élaborer un alphabet personnel dans n'importe quel ordre. Cela est particulièrement pratique lorsqu'un programme doit servir à manipuler du texte codé en TTS pour être employé avec un équipement de photocomposition. Ce code (différent des codes ASCII ou EBCDIC) a été conçu avant l'apparition des ordinateurs. Les codes numériques qui représentent les lettres de l'alphabet sont déterminés par l'ordre de fréquence suivant lequel apparaissent les caractères, et non par ordre alphabétique. A cette époque, le classement par ordre alphabétique était impossible. En Turbo, le programmeur définirait simplement un type de données appelé CodeTTS et listerait les codes par ordre alphabétique.

Le Turbo propose également un jeu d'outils puissants pour combiner les éléments de données de types différents en structures cohérentes appelées *enregistrements*. Tous ces points seront abordés ultérieurement.

OPÉRATEURS MATHÉMATIQUES ET ORDRE DE PRIORITÉ

La plupart des règles du Turbo relatives aux opérateurs mathématiques et à leur ordre d'exécution sont communes à presque tous les langages de programmation et les habitudes prises en BASIC ou en FORTRAN sont toujours valables. En Turbo, il faut prendre garde à l'incompatibilité de certains types de données, par exemple lors de l'addition de nombres réels et de nombres entiers ou de la division d'entiers si l'on veut obtenir comme résultat un nombre réel. Un mélange de différents types arrive parfois, mais là encore les messages d'erreur du Turbo signalent généralement le conflit. La Figure 2.9 résume les opérateurs du Turbo et les types de données qui peuvent leur être associés. La Figure 2.10 donne l'ordre de priorité des différents opérateurs.

Lors de l'écriture d'un programme, l'utilisateur doit tout d'abord faire usage de son bon sens et se référer aux tables lorsqu'une solution ne peut être rapidement trouvée par le système essai/erreur qui a été abordé précédemment.

Lorsque des nombres de type entier (INTEGER) et de type réel (REAL) sont simultanément utilisés dans des calculs, le résultat est de type réel.

Calcul	Type des opérandes	Type du résultat
* multiplication	entier	entier
	réel	réel
	entier, réel	réel
/ division	entier	réel
	réel	réel
	entier, réel	réel
+ addition	entier	entier
	réel	réel
	entier, réel	réel
- soustraction ou moins	entier	entier
	réel	réel
	entier, réel	réel

Les opérateurs suivants sont exclusivement réservés aux nombres entiers :

DIV	entier	entier
MOD	entier	entier

Les opérateurs suivants peuvent être associés à n'importe quel type de nombre, ils génèrent un résultat booléen (VRAI ou FAUX, 1 ou 0) :

=	égalité	<=	inférieur ou égal à
<	inférieur à	>=	supérieur ou égal à
>	supérieur à	<>	différent de

Les opérateurs suivants requièrent des valeurs booléennes ou entières :

NOT	arithmétique	entier	entier
	logique	booléen	booléen

Figure 2.9 : Opérateurs Turbo.

AND	arithmétique	entier	entier
	logique	booléen	booléen
OR	arithmétique	entier	entier
	logique	booléen	booléen

Les opérateurs suivants sont principalement destinés à manipuler des données exprimées sous forme binaire (bits) :

XOR	arithmétique	entier	entier
	logique	booléen	booléen
SHL	déplacement à gauche	entier	entier
SHR	déplacement à droite	entier	entier

Figure 2.9 (suite)

Les opérations sont exécutées de la gauche vers la droite ; si des opérateurs ont le même niveau de priorité, c'est celui qui est placé le plus à gauche qui est pris en compte en premier.

Les expressions placées entre parenthèses sont évaluées en premier (les ordres de priorités sont conservés à l'intérieur); si des parenthèses sont imbriquées, les expressions les plus internes sont prises en compte les premières.

En Pascal, l'ordre de priorité est le suivant :

Unaire moins	(-)
Négation logique	(NOT)
Toutes formes de multiplication et division	(*, /, DIV, MOD, AND, SHL, SHR)
Toutes formes d'addition et soustraction	(+, -, OR, XOR)
Opérateurs relationnels	(=, <, >, <>, <=, >=, IN)

Figure 2.10 : Ordre de priorité des opérateurs.

EXPRESSIONS

On a remarqué que les types de données ne sont pas suffisants pour résoudre les problèmes ; ils doivent être manipulés et combinés sous la forme d'*expressions*. Une expression est simplement une séquence de termes significative dont les différents éléments sont séparés par des opérateurs. En voici quelques exemples :

```
+2
Capital + Interet
3.14 * 208
Chiffres AND Lettres
7 + SQRT (144)
- Temperature
A < B
```

Une expression peut se composer de variables, de constantes et de fonctions. Quelques règles régissent l'emploi des expressions. La plus importante est la suivante : tous les opérateurs employés dans une expression doivent être autorisés pour les types de données auxquels ils se réfèrent. L'exemple du paragraphe relatif aux entiers et concernant les opérateurs DIV et MOD illustre cette règle. L'expression suivante n'est pas valide :

```
3.00 DIV 1.00
```

Bien que la réponse doive être 3, l'opérateur DIV ne peut être associé qu'à des entiers. Le compilateur Turbo signale ce type d'erreur, ce qui est un excellent moyen de faire assimiler les règles à l'utilisateur.

INSTRUCTIONS

Une instruction est un ordre quelconque, unique et clairement défini. Les instructions permettent l'accomplissement d'une tâche. Dans notre premier programme, la ligne :

```
WRITELN ('Salut !')
```

était une instruction. Dans le programme Tickets_de_bus, les lignes :

```
WRITELN ('Nombre de membres dans 1 amicale ?') ;
READLN (Membres) ;
```

étaient des instructions.
Les instructions se chargent de l'entrée et de la sortie ; elles prennent également le résultat de certaines instructions et l'assignent à un identificateur. Voici quelques exemples :

```
Tickets := Jeux * Membres
PrixTotal := Cout + Charges
Location := Loyer_Chambre
Pommes := Oranges
Somme_des_carres := SQR(A) + SQR(B)
Compteur := Compteur + 1
```

On remarque que les expressions doivent être combinées en instructions pour avoir un effet quelconque. Dans l'exemple suivant, l'expression 2 + 2 n'a aucun intérêt si le résultat n'est pas exploité :

```
Somme := 2 + 2
```

On remarque que l'*opérateur d'assignation* (:=) remplace le signe égal (=) habituel. Le Turbo utilise le signe égal uniquement pour tester l'équivalence. Un signe égal seul est considéré comme un opérateur relationnel. L'opérateur d'assignation attribue une certaine valeur à un identificateur. La définition d'une instruction d'assignation est la suivante : une variable et une expression liées par un opérateur d'assignation.

Le Turbo permet l'emploi d'un groupe d'instructions *(instructions composées)* chaque fois qu'une instruction unique est autorisée. Cette possibilité supprime la nécessité de constructions GOTO et permet l'élaboration de quelques structures très puissantes. Les instructions composées sont toujours encadrées par les mots réservés BEGIN et END. Dans le programme de la Figure 2.11, toutes les lignes placées entre ces mots forment une instruction composée. Il n'est pas nécessaire de s'occuper pour l'instant de l'instruction CASE ; celle-ci sera traitée dans le Chapitre 3 avec la création de procédures. Il faut simplement remarquer dans cette figure que les lignes multiples de l'instruction CASE remplacent

une ligne unique qui appelle la procédure appropriée. Au lieu d'un simple appel de procédure, l'instruction CASE (instruction composée comprenant 11 lignes) sélectionne l'une des 9 options différentes en fonction du caractère entré. Le résultat de 11 lignes de code (appelant une procédure) est identique à celui produit par une ligne unique.

La Figure 2.12 présente un exemple plus complexe d'instructions composées imbriquées.

```
PROCEDURE Interprete_choix;          {Interprétation d'un menu de sélection}

BEGIN
   READLN (Option);
   CASE Option OF
      1: Accepte_Nouv_Noms;
      2: Liste_Alphabetique;
      3: Tri_Etiquettes_Code_postal;
      4: Affiche_Liste_Sur_Ecran;
      5: Modification_Une_Entree;
      6: Creation_Liste_Adresses;
      7: Sortie;
   ELSE Gestion_erreur;
   END;
END;
```

Figure 2.11 : Utilisation d'instructions composées.

```
PROCEDURE Tri_Prog;
BEGIN                                         {Programme de tri}
   FOR Passes := Max_Enreg DOWNTO 2 DO
      BEGIN
         Intervert := TRUE;
         IF Intervert THEN
            BEGIN                             {Tant que Intervert est vrai}
               I := 1;
               Intervert := FALSE;
               WHILE I < Max_Enreg DO
                  BEGIN                       {Interversion}
                     IF Tri[I].Nom > Tri[I+1].Nom THEN Change;
                     I := I+1;
                  END;                        {Fin d'interversion}
            END;                              {Tant que Intervert est vrai}
         END;                                 {Chaque passe}
END;                                          {Procédure de tri}
```

Figure 2.12 : Programme avec instructions composées imbriquées.

Cet exemple de programme, qui peut sembler un peu complexe au premier abord, peut être fragmenté en différentes parties encadrées d'instructions BEGIN et END. On remarque l'emploi de commentaires pour identifier les couples BEGIN et END. Cet exemple montre également comment l'indentation aide à identifier les instructions composées comme des modules logiques.

3. OUTILS ET TECHNIQUES COURANTS

INTRODUCTION ET AVERTISSEMENT RELATIF À L'INCOMPATIBILITÉ

Tous les programmes ont besoin de manipuler des informations en provenance de l'extérieur. Un programme totalement autonome est par définition sans intérêt. La plupart des programmes attendent une interaction humaine généralement par l'intermédiaire du clavier et de l'écran. De nombreux programmes envoient également des résultats à une imprimante ; les plus sophistiqués à d'autres ports tels qu'un modem, une manette de jeu, une table traçante, un ensemble de photocomposition, etc. Lorsque les programmeurs parlent d'entrée/sortie, ce sont les éléments qui leur viennent à l'esprit.

L'autre opération d'entrée/sortie concerne l'écriture ou la lecture de fichiers. Le Pascal standard n'était pas très performant dans ce domaine et n'offrait pratiquement aucune aide pour les utilisations interactives les plus banales du clavier et de l'écran. Toutes ces opérations passaient par les cartes perforées ou les bandes magnétiques auxquelles on accédait séquentiellement, élément par élément. Les fichiers à accès aléatoire ne faisaient pas partie du Pascal initial. Les écrans interactifs (sans parler de fenêtres, ni de couleur, ni de graphisme) étaient inconnus. A cause de ces restrictions sévères, chaque implémentation du Pascal offre quelques extensions d'entrée/sortie particulières. Le Turbo possède une panoplie exceptionnelle d'outils d'entrée/sortie qui ne se

contentent pas de surpasser ceux de tous les autres Pascal mais offrent des possibilités inégalées par tous les autres langages disponibles sur des micro-ordinateurs ; seul un langage d'assemblage offre des performances identiques.

Alors que les techniques d'adressage de port et de gestion d'interruptions sont des caractéristiques sophistiquées, chaque programme contient des instructions d'entrée/sortie. Nous verrons que le Turbo est très performant dans sa façon de manipuler les entrées/sorties, quel que soit le périphérique employé. Mais il faut savoir que tous ces avantages se font aux dépens de la compatibilité avec le Pascal standard et la plupart des autres Pascal. Presque tous les aspects de l'entrée/sortie du Turbo, depuis l'interaction écran élémentaire jusqu'au transfert de fichiers disque importants, lui sont spécifiques ou diffèrent d'une certaine façon des autres implémentations.

INTERACTION AVEC L'UTILISATEUR

La Figure 3.1 présente un programme appelé Simple qui illustre les entrées/sorties sur un écran de base. On peut essayer de l'exécuter après avoir mis l'imprimante sous tension.

La section de déclaration contient seulement trois identificateurs (PremEntier, DeuxEntier et Somme) qui sont des entiers. Il n'y a pas de procédure ni de fonction ; le corps du programme se compose de huit instructions encadrées par les mots réservés BEGIN et END.

Les entrées du programme se composent de deux entiers tapés par l'utilisateur. Le résultat ne comporte pas uniquement la somme finale mais également les messages et les signaux que le programme affiche sur l'écran pendant son déroulement. On pense souvent que seules les dernières lignes d'un programme génèrent des résultats utiles, mais le Turbo considère toutes les informations écrites par le programme comme des résultats.

On remarque que les résultats sont envoyés sur l'écran ainsi que sur l'imprimante. En effet, à l'inverse de certaines versions du BASIC, le Turbo n'a pas de commandes PRINT et LPRINT distinctes. En fait, il emploie un jeu de procédures standard pour la lecture et l'écriture par l'intermédiaire des écrans, des claviers, des imprimantes, des modems et des autres périphériques ainsi que des fichiers texte et données. Comme le Pascal initial définissait son environnement en termes de fichiers séquentiels, le Turbo

définit aussi l'environnement d'entrée/sortie comme étant la lecture et l'écriture de fichiers. Il existe cependant une différence essentielle : le Turbo définit un fichier d'une façon beaucoup plus souple pour inclure les périphériques cités plus haut.

On exécute le programme Simple en entrant 5 et 9 en réponse aux signaux ; l'ordinateur affiche alors sur l'écran les informations de la Figure 3.2. Il sort simultanément sur l'imprimante :

La somme est : 14

La procédure WRITELN prédéfinie prend en charge la sortie ; de même, la procédure READLN prend en charge la lecture de deux nombres tapés en réponse aux signaux et leur stockage dans les variables PremEntier et DeuxEntier.

Nous allons examiner plus attentivement la première instruction :

WRITELN (OUTPUT, 'Ce programme additionne deux entiers') ;

Toutes les entrées/sorties du Turbo ont un format similaire. La commande (READ, READLN, WRITE ou WRITELN) est suivie d'un argument entre parenthèses. L'argument spécifie deux choses : le fichier à lire ou à écrire et les données concernées. Dans cet exemple, nous écrivons une ligne dans un fichier. Le nom du fichier est OUTPUT et les données correspondent au message placé entre apostrophes :

'Ce programme additionne deux entiers'

Les apostrophes, comme les guillemets d'une instruction PRINT BASIC, indiquent simplement au Turbo que l'information est de type texte et qu'il ne s'agit pas d'un identificateur (c'est-à-dire un nom de variable). Lors de l'impression de nombres, les commandes d'écriture peuvent avoir des arguments supplémentaires pour contrôler le positionnement horizontal. Par exemple, on peut spécifier le nombre d'espaces de tête précédant la valeur, la largeur du champ qu'elle peut occuper et, avec les nombres réels, le nombre de chiffres apparaissant après le point décimal.

Dans cet exemple, OUTPUT est un identificateur qui se réfère à un *fichier de sortie standard* prédéfini. Sauf indication contraire donnée au système, ce fichier est l'écran de l'ordinateur. De même, il existe un *fichier d'entrée standard* prédéfini appelé INPUT ; de façon identique, sauf indication contraire, il s'agit du clavier de l'ordinateur. Nous allons examiner maintenant l'instruction WRITELN suivante :

WRITELN ('Quel est le deuxieme nombre ?') ;

Il semble que quelque chose manque, mais le programme se déroule quand même. Si aucun nom de fichier n'est donné avec une instruction d'écriture (sortie), le Turbo suppose que les résultats doivent être dirigés sur l'écran. De même, dans une instruction de lecture (entrée), le Turbo suppose que celle-ci doit être effectuée à partir du clavier si aucun nom de fichier n'est associé à la commande. La dernière instruction WRITELN est :

WRITELN (LST,'La somme est :',Somme)

Dans ce cas, les données à sortir correspondent au message (La somme est :) et le nom de fichier (LST) est l'identificateur standard désignant presque toujours une imprimante.

```
PROGRAM Simple;

VAR
  Prem_Entier, Deux_Entier, Somme : INTEGER;

BEGIN
   WRITELN (OUTPUT, 'Ce programme additionne deux entiers');
   WRITELN ('Tapez le premier nombre :');
   READLN (INPUT, Prem_Entier);
   WRITELN ('Tapez le second nombre :');
   READLN (Deux_Entier);
   Somme := Prem_Entier + Deux_entier;
   WRITELN ('La somme des deux nombres est ',Somme);
   (*WRITELN (LST, 'LA SOMME EST : ',Somme)*)
END.
```

Figure 3.1 : Programme effectuant des entrées/sorties.

```
Ce programme additionne deux entiers
Tapez le premier nombre :
54
Tapez le second nombre :
53
La somme des deux nombres est 107
```

Figure 3.2 : Sortie de résultat.

Fichiers standard

Avec ce format simple, le Turbo réalise virtuellement toutes ses opérations d'entrée/sortie. Les données de sortie sont des caractères, des nombres, des variables, des constantes, des chaînes et des enregistrements entiers. Les fichiers peuvent être des fichiers disque ou des périphériques physiques. Pour les distinguer, le Turbo possède plusieurs identificateurs standard pour les fichiers associés à des périphériques physiques courants. La liste de ces identificateurs apparaît Figure 3.3. En général, avant de permettre la lecture et l'écriture de fichiers, le Turbo requiert quelques étapes préalables pour identifier le fichier, l'ouvrir, placer le pointeur au début du fichier puis le fermer lorsque ces opérations sont terminées. Le Turbo exécute automatiquement ces tâches lorsque l'on utilise l'un des fichiers standard prédéfinis qui apparaissent dans la Figure 3.3. En effet, cela signifie que même si les périphériques physiques sont traités comme des fichiers lors de l'emploi de commandes READ et WRITE, les désagréments sont épargnés à l'utilisateur.

Les fichiers de ce type les plus importants sont INPUT, OUTPUT et LST. Ils prennent en charge toutes les opérations d'entrée/sortie ordinaires ainsi que les opérations d'impression. Le Turbo donne simplement des *noms de fichiers logiques* aux données en les dirigeant vers les périphériques physiques (ou en les recevant). Cela signifie qu'il est possible d'écrire un programme Turbo sans tenir compte des périphériques physiques associés au système. Les noms de périphériques logiques constituent un outil particulièrement puissant pour l'écriture de programmes de communication ou de programmes gérant des machines telles qu'une photocomposeuse, un panneau d'affichage électronique, etc. Par exemple, on peut écrire puis mettre au point un programme de communication en utilisant la console comme périphérique physique, auquel on attribue un nom logique identique à celui du modem. On peut surveiller les résultats sur l'écran pendant la mise au point du code initial puis assigner au modem le nom de fichier OUTPUT uniquement lors de la mise au point du code de contrôle de la communication.

Fichiers standard et périphériques logiques

Il peut y avoir confusion entre les fichiers texte standard CON (console), TRM (terminal) et KBD (*keyboard* : clavier). Le fichier CON dirige le résultat sur l'écran, prend des informations à partir

du clavier et en renvoie un écho sur l'écran. Il sert également de mémoire tampon pour les entrées, ce qui permet de revenir en arrière et d'effacer des caractères avant de transmettre les données à l'aide de la touche Return. Son effet est identique à celui des fichiers par défaut, INPUT et OUTPUT.

Fichiers par défaut, sans affectation supplémentaire :

INPUT	Fichier d'"entrée" standard, associé aux entrées à partir du clavier. Les caractères tapés sont stockés dans une zone tampon, ce qui permet de corriger les erreurs avec les touches Backspace et Escape. Un écho est généré sur l'écran, il inclut les caractères "retour chariot" et "saut de ligne".
OUTPUT	Fichier de "sortie" standard, associé normalement à l'écran ; il prend en compte les caractères "retour chariot" et "saut de ligne".

Fichiers utilisés en entrée et en sortie :

CON	"Console" réalise les entrées à partir du clavier et les sorties sur l'écran. Les fonctions sont identiques à celles utilisées avec INPUT et OUTPUT.
TRM	"Terminal" réalise les entrées à partir du clavier et les sorties sur l'écran ; un écho est associé aux caractères tapés à partir du clavier mais ils ne sont pas stockés dans une zone tampon. Il est donc impossible de corriger les caractères erronés ; il n'est pas nécessaire de taper la touche Return pour valider une entrée.
AUX	"Auxiliaire" réalise les entrées et les sorties. Il est normalement associé à un modem mais peut se référer à d'autres périphériques comme un lecteur de bandes perforées ou un perforateur de bandes.
USR	"Utilisateur" réalise les entrées et les sorties ; il est seulement utilisé dans des programmes élaborés et est associé à des modules de gestions d'entrées/sorties.

Figure 3.3 : Identificateurs standard.

```
Fichiers utilisés en entrée ou en sortie :

KBD     "Clavier" réalise seulement les entrées ; les caractères ne
        sont pas stockés dans une zone tampon ni renvoyés en écho.

LST     "Listing" réalise seulement les sorties ; elles sont
        généralement dirigées sur l'imprimante. La sortie étant gérée
        par le système d'exploitation, il n'est pas nécessaire de
        savoir si l'imprimante est connectée au port série ou au port
        parallèle.
```

Figure 3.3 (suite)

Bien que le simple remplacement de INPUT par KBD et de OUTPUT par TRM semble suffire pour obtenir un fonctionnement équivalent, ce n'est pas le cas. INPUT et CON font un excellent usage de la mémoire tampon associée au clavier pour rendre les programmes plus tolérants vis-à-vis des erreurs de frappe. Lors d'entrées au clavier, l'écran renvoie un écho des touches tapées, mais celles-ci ne sont pas immédiatement intégrées au programme ; elles sont placées dans une zone de stockage temporaire appelée *mémoire tampon*. Dans cette zone, on peut manipuler les caractères en exécutant par exemple une insertion, un effacement ou une réécriture. Il faut taper la touche Return pour envoyer les caractères de cette zone dans le programme.

Cette procédure est utilisée dans les machines à écrire électroniques qui possèdent une petite zone tampon pour permettre à l'utilisateur de corriger ses erreurs de frappe avant que les caractères apparaissent sur le papier. La zone tampon est fournie par le système d'exploitation et non par le logiciel. Le fichier KBD ne possède pas de zone tampon et les entrées n'ont pas d'écho sur l'écran (cette caractéristique présente certains intérêts ; par exemple, lorsqu'un mot de passe ne doit pas apparaître sur l'écran). Pour visualiser l'effet de l'utilisation de ces noms de fichiers, on modifie le programme de la Figure 3.1 pour qu'il ressemble à celui de la Figure 3.4.

```
PROGRAM Simple;

VAR
  Prem_Entier, Deux_Entier, Somme : INTEGER;

BEGIN
   WRITELN (OUTPUT, 'Ce programme additionne deux entiers');
   WRITELN ('Tapez le premier nombre :');
   READLN (KBD, Prem_Entier);          {Comment l'écran est-il affecté ?}
   WRITELN ('Tapez le second nombre :');
   READLN (CON, Deux_Entier);          {Comment l'écran est-il affecté ?}
   Somme := Prem_Entier + Deux_entier;
   WRITELN ('La somme des deux nombres est ',Somme);
   WRITELN (LST, 'LA SOMME EST : ',Somme)·
END.
```

Figure 3.4 : Programme modifié pour illustrer l'utilisation d'autres fichiers d'E/S standard.

Les deux autres fichiers prédéfinis sont AUX et USR ; ils représentent les périphériques AUXiliaire et UtiliSateuR. La plupart du temps, AUX sert à se référer au port RS232C de l'ordinateur, particulièrement lorsque les données traitées passent par un modem. Le fichier USR est généralement employé dans les programmes d'entrée/sortie sophistiqués écrits pour piloter un appareil tel qu'un robot, une table traçante ou un affichage graphique particulier.

Les programmeurs familiarisés avec le système d'exploitation CP/M trouveront de sérieuses ressemblances entre les noms de fichiers standard du Turbo et ceux que ce système d'exploitation attribue aux périphériques logiques. Cette similitude peut prêter à confusion parce que le Turbo emploie également la notion de périphérique logique. La liste des périphériques logiques du Turbo apparaît Figure 3.5. La plupart de ces noms ressemblent aux identificateurs de fichiers standard mais les périphériques logiques se terminent toujours par deux-points. Dans la programmation courante, on peut se contenter d'employer le nom de fichier standard tel que LST et ignorer la distinction entre ce nom et le nom de périphérique logique LST.

Dans les exemples de programmes étudiés jusqu'à présent, nous avons employé les fonctions WRITELN et READLN (écrire une ligne et lire une ligne). Lorsque l'on ne veut pas écrire ou lire toutes les informations d'une ligne (c'est-à-dire jusqu'à un marqueur de fin de ligne), le Turbo propose deux autres commandes : WRITE

Les organes suivants sont traités comme des fichiers texte lors des opérations d'entrée/sortie. On remarque qu'à chacun des nom est associé un nom de fichier prédéfini (voir la Figure 3.3).

Organes logiques utilisés en entrée et en sortie :

CON — Console réalise les entrées à partir du clavier et les sorties sur l'écran. Un écho est généré sur l'écran après la frappe de chaque caractère. Les caractères tapés sont stockés dans une zone tampon d'une ligne permettant à l'utilisateur de corriger les erreurs avant validation de l'entrée.

TRM — Terminal réalise les entrées à partir du clavier et les sorties sur l'écran ; un écho est associé aux caractères tapés à partir du clavier mais ils ne sont pas stockés dans une zone tampon.

AUX — Auxiliaire réalise les entrées et les sorties. Il est normalement associé à un modem mais peut se référer à d'autres périphériques comme un lecteur de bandes perforées ou un perforateur de bandes. Avec CP/M, AUX est associé aux organes logiques RDR: (lecteur) et PUN: (perforateur).

USR — Utilisateur réalise les entrées et les sorties ; il est seulement utilisé dans des programmes élaboré et est associé à des modules de gestions d'entrées/sorties. Par exemple, un contrôle de robot, un système de photo-composition ou une communication en temps réel avec un échantillonneur de données.

Organes logiques utilisés en entrée ou en sortie :

KBD — Clavier réalise seulement les entrées ; les caractères ne sont pas stockés dans une zone tampon ni renvoyés en écho.

LST — Listing réalise seulement les sorties ; elles sont généralement dirigées sur l'imprimante. La sortie étant gérée par le système d'exploitation, il n'est pas nécessaire de savoir si l'imprimante est connectée au port série ou au port parallèle.

Figure 3.5 : Périphériques logiques.

et READ. Nous examinerons rapidement la différence entre ces deux groupes de commandes, mais certaines variations des commandes READLN et WRITELN sont illustrées dans la Figure 3.6.

On entre puis on exécute le programme de cette figure. A priori, il génère un résultat identique à celui du programme précédent, mais pour plus de simplicité la ligne relative à l'imprimante a été supprimée. Il comporte d'autres modifications mineures concernant la manière dont les données sont manipulées. Tout d'abord, la séquence du premier programme :

```
WRITELN ('Quel est le premier nombre ?) ;
READLN (INPUT, PremEntier) ;
WRITELN ('Quel est ıe second nombre ?') ;
READLN (DeuxEntier) ;
```

a été remplacée par les lignes suivantes, qui sont plus compactes :

```
WRITELN (Tapez les deux nombres a additionner.') ;
READLN (PremEntier, DeuxEntier) ;
```

Un résultat classique du premier programme peut être :

```
La somme des deux nombres est 278
```

alors que le résultat du second peut se présenter sous la forme :

```
La somme des deux nombres est
     278
```

Ces distinctions ne sont pas d'une importance capitale dans des programmes d'une telle simplicité. Cependant, lors de la lecture de fichiers importants, ce type de procédure évite d'écrire des structures peu maniables destinées à segmenter des blocs de données en partie significatives (par exemple, nom, adresse, ville et code postal).

Les commandes READLN et READ peuvent lire un nombre illimité de variables à condition qu'un délimiteur de fin de ligne ne sépare pas les valeurs. On voit cela dans le second programme qui minimise l'emploi du signal demandant une entrée. Dans ce programme, il suffit d'indiquer à l'instruction READLN les identi-

ficateurs à employer pour le stockage des valeurs entrées. Les commandes sont assez souples et, si les identificateurs ont été déclarés au préalable, une instruction READ peut accepter une série de valeurs de types différents.

Nous verrons des exemples de programmes qui emploient les commandes READ et READLN de manière apparemment interchangeable. Ce qui les différencie est un point assez confus ; cette différence n'est pas évidente et n'a malheureusement pas été traitée dans la documentation standard. Comme les autres commandes, les opérations READ et READLN sont logiques et efficaces lorsque l'on a bien compris leur rôle. On exécute la série de programmes, puis on examine les résultats des Figures 3.7 à 3.16. Elles devraient éclairer l'utilisateur sur ce qui est en fait une opération très simple.

```
PROGRAM Ecriture_lignes;

VAR
  Prem_Entier, Deux_Entier, Somme : INTEGER;

BEGIN
   WRITELN ('Ce programme additionne deux entiers et illustre quelques nouvelles techniques');
   WRITELN ('Tapez les deux nombres à additionner :');
   READLN (Prem_Entier, Deux_Entier);
   Somme := Prem_Entier + Deux_Entier;
   WRITELN;
   WRITELN ('La somme des deux nombres est ',Somme);
END.
```

Figure 3.6 : Programme illustrant le compactage des lignes d'entrée.

On exécute le programme de la Figure 3.7. La Figure 3.8 montre le résultat obtenu sans entrer de données (juste avec deux frappes de la touche Return). Comme on pouvait s'y attendre, les variables sont égales à leur valeur d'initialisation, 0.

La Figure 3.9 montre l'écran d'affichage avec des données autorisées entrées sur une seule ligne. Dans tous ces affichages, les nombres sont séparés par des espaces.

```
PROGRAM Illustration_de_READ_et_READLN;

VAR
   A, B, C, D : INTEGER;

PROCEDURE Initialise_valeurs;
BEGIN
   A := 0;
   B := 0;
   C := 0;
   D := 0;
END;

PROCEDURE Essais;
BEGIN
   WRITELN; WRITELN;                    {pour clarifier l'écran}
   WRITELN ('A = ', A);
   WRITELN ('B = ', B);
   WRITELN ('C = ', C);
   WRITELN ('D = ', D);
END;

BEGIN
   Initialise_valeurs;
   WRITELN ('Tapez 4 nombres entiers : ');
   WRITELN;
   READLN (A, B);
   READ (C, D);
   Essais
END.
```

Figure 3.7 : Démonstration de l'utilisation des commandes READ et READLN.

On remarque que le programme a déjà lu les deux premières valeurs de la ligne bien qu'il recherche quatre nombres. L'instruction READLN lui indique qu'après la lecture de A et de B il doit ignorer toutes les autres entrées de la ligne.

```
Tapez 4 nombres entiers :

←┘
←┘

A = 0
B = 0
C = 0
D = 0
```

Figure 3.8 : Résultat du déroulement du programme de la Figure 3.7, sans entrée de valeurs.

```
Tapez 4 nombres entiers :

1  2  3  4 ←┘
←┘

A = 1
B = 2
C = 0
D = 0
```

Figure 3.9 : Résultat du déroulement du programme de la Figure 3.7, avec entrée de valeurs tapées sur une seule ligne.

La Figure 3.10 montre le résultat obtenu avec des données valides entrées sur deux lignes.

```
Tapez 4 nombres entiers :

1  2  3  4 ←┘
5  6  7  8 ←┘

A = 1
B = 2
C = 5
D = 6
```

Figure 3.10 : Résultat du programme de la Figure 3.7 avec l'entrée des valeurs tapées sur deux lignes.

De nouveau, on a indiqué au système de lire les deux premières valeurs de la première ligne et les deux premières valeurs de la seconde ligne (nous avons tapé plus de valeurs que nécessaire pour montrer exactement ce qui est lu et ce qui est ignoré).

La Figure 3.11 montre le même programme que celui de la Figure 3.7, mais avec la première instruction READLN remplacée par une

```
PROGRAM Illustration_de_READ_et_READLN;

VAR
   A, B, C, D : INTEGER;

PROCEDURE Initialise_valeurs;
BEGIN
   A := 0;
   B := 0;
   C := 0;
   D := 0;
END;

PROCEDURE Essais;
BEGIN
   WRITELN; WRITELN;                  {pour clarifier l'écran}
   WRITELN ('A = ', A);
   WRITELN ('B = ', B);
   WRITELN ('C = ', C);
   WRITELN ('D = ', D);
END;

BEGIN
   Initialise_valeurs;
   WRITELN ('Tapez 4 nombres entiers : ');
   WRITELN;
   READ (A, B);                       {modification de READLN en READ}
   READ (C, D);
   Essais
END.
```

Figure 3.11 : Programme modifié pour illustrer la commande READ.

instruction READ. Cette petite modification a des effets surprenants. La Figure 3.12 montre le résultat généré par ce programme. Elle ne comporte pas d'erreurs ; les nombres ont tous été tapés suivis d'un espace. Après le chiffre 9, un retour chariot (touche Return) ; *l'écran n'affiche aucune indication de cette opération car l'instruction READ ne tient pas compte des marqueurs de fin de ligne.* En regardant l'écran, on peut se demander pourquoi le programme semble avoir sauté aléatoirement de 2 à 10. La procédure suivie sera plus claire en rendant visible le retour chariot. La Figure 3.13 montre l'écran avec le retour chariot affiché.

```
Tapez 4 nombres entiers :

1 2 3 4 5 6 7 8 9 10 11 12 13 14


A = 1
B = 2
C = 0
D = 0
```

Figure 3.12 : Résultat du programme modifié avec entrée des valeurs tapées sur deux lignes différentes.

```
Tapez 4 nombres entiers :

1 2 3 4 5 6 7 8 9←┘10 11 12 13 14


A = 1
B = 2
C = 10
D = 11
```

Figure 3.13 : Affichage de la Figure 3.12 avec représentation du retour chariot.

La Figure 3.14 illustre une autre situation telle qu'elle devrait apparaître sur l'écran. La Figure 3.15 montre le même affichage avec représentation des retours chariot.

```
Tapez 4 nombres entiers :

1       2


A = 1
B = 0
C = 2
D = 0
```

Figure 3.14 : Résultat du programme modifié avec deux valeurs séparées par un retour chariot non représenté.

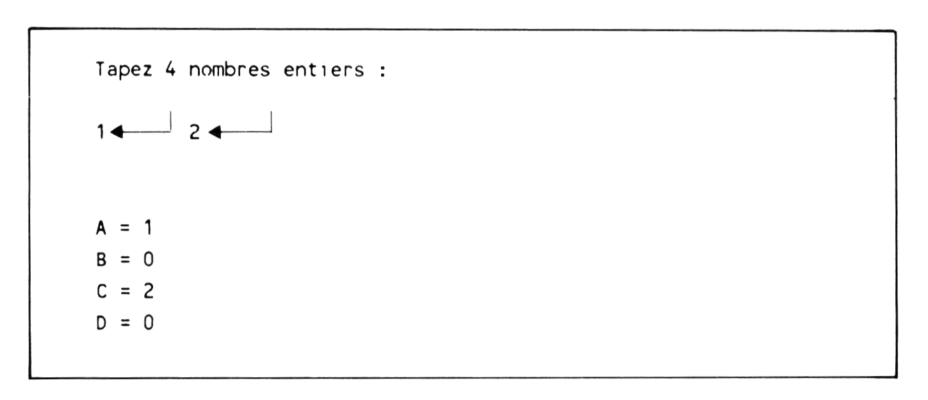


Figure 3.15 : Affichage de la Figure 3.14 avec représentation des retours chariot.

Enfin, la Figure 3.16 montre le programme à nouveau modifié et la Figure 3.17 son résultat.

```
PROGRAM Illustration_de_READ_et_READLN;

VAR
   A, B, C, D : INTEGER;

PROCEDURE Initialise_valeurs;
BEGIN
   A := 0;
   B := 0;
   C := 0;
   D := 0;
END;

PROCEDURE Essais;
BEGIN
   WRITELN; WRITELN;                  {pour clarifier l'écran}
   WRITELN ('A = ', A);
   WRITELN ('B = ', B);
   WRITELN ('C = ', C);
   WRITELN ('D = ', D);
END;

BEGIN
   Initialise_valeurs;
   WRITELN ('Tapez 4 nombres entiers : ');
   WRITELN;
   READLN (A, B, C, D);               {toutes les données sur 1 ligne}
   Essais
END.
```

Figure 3.16 : Programme modifié pour rechercher toutes les valeurs sur une seule ligne.

```
Tapez 4 nombres entiers :

1 2 3 4 5 6 7 8 9 ←┘
A = 1
B = 2
C = 3
D = 4
```

Figure 3.17 : Résultat de l'exécution du programme modifié avec entrée des valeurs tapées sur une seule ligne.

C'est peut-être la caractéristique la plus obscure du Turbo. Il est conseillé d'observer à nouveau les exemples si la distinction entre les instructions READ et READLN n'est pas correctement assimilée ou encore mieux, de modifier les programmes et de faire de nouveaux essais.

Pour résumer le déroulement de ces commandes lorsqu'elles lisent des entrées à partir d'un fichier de données, la commande READLN accepte toutes les valeurs recherchées puis ignore toute entrée ultérieure effectuée sur la même ligne ; une nouvelle lecture sera effectuée au début de la ligne suivante. Inversement, la commande READ s'arrête lorsqu'elle a trouvé toutes les données recherchées ; ensuite elle reste sur la même ligne pour procéder à l'opération de lecture suivante.

On observe à nouveau le programme de la Figure 3.1 pour voir comment le Turbo écrit des résultats. Les commandes WRITELN et WRITE ont des effets différents. WRITELN termine toujours son affichage par une séquence de fin de ligne. WRITE effectue son affichage et ne "bouge " plus. Par conséquent, une instruction WRITELN seule génère l'affichage d'une ligne vierge. Dans le programme de la Figure 3.4, la valeur Somme a été affichée sur une ligne à part parce que le message :

La somme des deux nombres est

était généré à l'aide de la commande WRITELN. On revient à ce programme pour effectuer les modifications suivantes. Remplacer :

```
WRITELN ('La somme des deux nombres est') ;
```

par :

```
WRITE ('La somme des deux nombres est') ;
```

puis observer l'effet obtenu.

Formatage du résultat

Tous les programmes étudiés jusqu'à présent ont seulement employé des entiers. Nous avons évité l'utilisation de nombres réels car ils nécessitent une manipulation particulière. Il s'agit là encore d'une fonction "oubliée" dans la documentation fournie avec le Turbo Pascal. Le Turbo présente une souplesse exceptionnelle

en ce qui concerne le formatage de résultats et cette caractéristique ne devrait pas être enterrée dans une liste obscure de paramètres associés à la fonction WRITE. Les quelques programmes suivants doivent permettre à l'utilisateur d'acquérir une certaine pratique concernant l'obtention de résultats présentés dans le format le plus clair possible.

Le programme appelé Surface_Cercle (Figure 3.18) utilise des nombres réels.

On entre le programme puis on l'exécute. Avec un rayon de 1234.56789, le résultat doit se présenter comme sur la Figure 3.19. On s'aperçoit que la notation exponentielle n'est pas exactement la présentation la plus claire.

```
PROGRAM Surface_Cercle;

VAR
   Rayon, Surface : REAL;

CONST
   Pi = 3.1416;

BEGIN
   WRITELN ('Rayon du cercle :');
   READLN (Rayon);
   Surface := Pi * Rayon * Rayon;
   WRITELN ('La surface d un cercle de rayon', Rayon, ' unités est : ');
   WRITELN (Surface, ' unités au carré')
END.
```

Figure 3.18 : Programme exécutant des calculs à partir de nombres réels.

```
Rayon du cercle :
1234.56789
La surface d un cercle de rayon  1.2345678900E+03 unités est :
  4.7882943801E+06 unités au carré
```

Figure 3.19 : Résultat du déroulement du programme Surface_Cercle, représentant une entrée/sortie de nombres réels non formatés.

La Figure 3.20 présente les techniques disponibles pour clarifier la présentation. On exécute à nouveau le programme ; si on entre un rayon de 1234.56789, le résultat doit être identique à celui de la Figure 3.21.

Le Turbo permet de passer sans difficulté de la notation exponentielle à la notation décimale conventionnelle, mais aussi de choisir le nombre de chiffres à placer après le point décimal, le nombre d'espaces précédant la valeur et même la justification à droite ou à gauche. De plus, il autorise ces manipulations sans affecter la manière dont est réalisé le stockage interne. Par conséquent, l'utilisateur est libre d'arrondir des chiffres pour l'impression d'un document officiel et de retenir les valeurs décimales pour une plus grande précision dans des calculs ultérieurs.

```
PROGRAM Sorties_formatees;

VAR
   Rayon, Surface : REAL;

CONST
   Pi = 3.1416;

BEGIN
   WRITELN ('Rayon du cercle :');
   READLN (Rayon);
   WRITELN ('Le rayon entré était :', Rayon);
   WRITELN ('Le rayon entré était :', Rayon:1:1);
   WRITELN ('Le rayon entré était :', Rayon:7:2);
   WRITELN ('Le rayon entré était :', Rayon:10:3);
   WRITELN ('Le rayon entré était :', Rayon:12:6);
   Surface := Pi * SQR(Rayon);
   WRITE ('La surface d un cercle de rayon', Rayon:11:5, ' unités est : ');
   WRITELN (Surface:15:6, ' unités au carré')
END.
```

Figure 3.20 : Programme de démonstration des options de formatage de résultats.

```
Rayon du cercle :
1234.56789
Le rayon entré était :  1.2345678900E+03
Le rayon entré était :1234.6
Le rayon entré était :1234.57
Le rayon entré était :  1234.568
Le rayon entré était : 1234.567890
La surface d un cercle de rayon 1234.56789 unités est : 4788294.380100 unités au carré
```

Figure 3.21 : Résultats formatés de différentes manières.

Les commandes WRITE et WRITELN étudiées auparavant utilisaient la syntaxe :

WRITELN (Fichier, Var1, Var2, ...)

Dans cette notation, *Fichier* signifie variable de fichier ou simplement le nom d'un fichier. *Var1, Var2,* etc. se réfèrent à des identificateurs de variables individuelles. Ceux-ci peuvent également être des constantes ou du texte. Lorsque l'on a tapé dans le premier exemple :

WRITELN (LST,'La somme est :',Somme)

le *fichier* était LST, représentant la sortie imprimante. La première variable était le texte encadré d'apostrophes et la deuxième, la variable entière Somme. On voit sur la Figure 3.21 qu'il est possible d'ajouter des informations à la commande pour indiquer non seulement ce qui doit être affiché, mais également comment l'affichage doit être présenté.
Lors de l'écriture de nombres réels, on utilise des paramètres supplémentaires après l'identificateur de variable pour indiquer au Turbo le nombre d'espaces précédant la valeur et le nombre de chiffres à afficher après le point décimal. Dans le dernier programme, on a tapé l'instruction :

WRITELN ('Le rayon entre etait :',Rayon :12 :6) ;

ayant pour résultat :

Le rayon entre etait : 1234.567890

L'ensemble du nombre occupe 12 caractères (dans cet exemple, il y a 10 chiffres, un point décimal et un espace de tête) et réserve 6 de ses espaces pour la partie décimale (.567890). Si on regarde les autres exemples de lignes, on remarque le comportement du Turbo : s'il n'y a pas assez de place pour représenter la partie entière, il ne la tronque pas mais l'affiche complètement.

Le Turbo permet également de contrôler la manière dont les entiers sont affichés. La Figure 3.22 est une version simplifiée de l'exemple de programme précédent.

La Figure 3.23 montre l'affichage obtenu après déroulement de ce programme. La variable A est affichée sans caractéristiques

particulières de format ; il n'y a pas d'espace de tête. Pour la variable B, on a indiqué au programme d'occuper 7 espaces et il a affiché 3 espaces de tête avant les 4 chiffres. Pour la variable C, on lui a demandé d'utiliser un seul espace mais cela était impossible ; le Turbo a ignoré cette spécification et utilisé le format par défaut. Pour la variable D, il a utilisé 6 espaces dont 2 espaces de tête.

```
PROGRAM Illustration_Impression_Entiers;

VAR
   A : INTEGER;

PROCEDURE Essais;
BEGIN
   WRITELN; WRITELN;                        {pour clarifier l'écran}
   WRITELN ('A = ', A);
   WRITELN ('B = ', A:7);
   WRITELN ('C = ', A:1);
   WRITELN ('D = ', A:6);
END;

BEGIN
   WRITELN ('Tapez 1 nombre entier : ');
   WRITELN;
   READLN (A);
   Essais
```

Figure 3.22 : Programme de démonstration des options de format pour l'impression d'entiers.

```
Tapez 1 nombre entier :

1234


A = 1234
B =    1234
C = 1234
D =   1234
```

Figure 3.23 : Résultat du programme de la Figure 3.22.

Le Turbo permet d'afficher des résultats à n'importe quel endroit de l'écran à l'aide de la commande GOTOXY (x,y) dans laquelle x représente un déplacement horizontal et y un déplacement vertical à partir de l'angle supérieur gauche. La gamme des x et des y dépend du type d'écran employé. GOTOXY est semblable à la commande LOCATE disponible dans certaines versions du BASIC.

Cependant, pour la sortie, le Turbo n'a pas de commande similaire à la fonction TAB(n) du BASIC pour diriger horizontalement le positionnement d'une opération PRINT ou LPRINT. Il faut pour cela écrire une procédure personnelle.

Le fait que le Turbo ne possède pas ce type de fonction est une des raisons de sa force. En effet, si l'interpréteur BASIC est un programme important et lent, c'est parce qu'il se compose d'une vaste gamme de commandes dont la plupart sont rarement (ou jamais) employées. Le Turbo a été conçu pour générer des programmes aussi concis et aussi efficaces que possible. La plupart des programmeurs Turbo (et presque toutes les applications commerciales obtenues à partir du Pascal ou de C) se constituent une bibliothèque de procédures relatives à des tâches particulières qui peuvent facilement être insérées dans des programmes lorsque cela est nécessaire, ou simplement ajoutées au moment de la compilation.

STRUCTURES DE CONTRÔLE DE PROGRAMMES

Note historique

La plupart des exemples de programmes rencontrés jusqu'à présent étaient relativement simples ; ils suivaient une progression linéaire du début jusqu'à la fin. Bien que cette structure soit adaptée à l'exécution de calculs simples et à des affichages, elle ne se prête pas à la réalisation de tâches plus importantes que celles qui sont proposées par une calculatrice de poche.

Turbo offre une variété de structures qui accroît énormément les possibilités d'un programme. Les instructions peuvent être exécutées dans l'ordre choisi et également à plusieurs reprises, ce qui donne au Turbo une nette supériorité. Mais sa plus grande

qualité est son aptitude à tester une condition et à choisir un déroulement d'action basé sur le résultat de ce test.

Quand les ordinateurs étaient uniquement programmés en langage machine, toutes les opérations relatives au contrôle du déroulement du programme étaient très complexes et absolument incompréhensibles pour les non-initiés.

Les programmeurs devaient se charger de l'introduction de valeurs spécifiques dans différents registres et de la réalisation de comparaisons. Ils devaient constamment lire et masquer un jeu d'indicateurs d'état et donner à l'ordinateur la conduite à suivre en fonction des résultats. Le choix d'une décision nécessitait souvent plusieurs comparaisons et impliquait toujours la sauvegarde et le chargement de données avant et après les tests, ce qui était fastidieux.

Les structures de contrôle de programmes sont encore ressenties comme des opérations difficiles, déconcertantes et ennuyeuses.

Heureusement, le Turbo offre une série de structures de contrôle souples, très puissantes et compréhensibles presque intuitivement. Les décisions les plus sophistiquées et les plus compliquées peuvent être codées en imbriquant des structures jusqu'à des niveaux de complexité difficiles à imaginer.

De plus, le Turbo offre un jeu de tests et de masques identiques à ceux qui sont disponibles en langage assembleur.

Avant d'explorer les structures spécifiques au Turbo, il ne faut pas oublier qu'à l'intérieur d'une structure de contrôle le Turbo autorise toujours l'utilisation d'une instruction composée comme celle d'une instruction simple. Cette caractéristique augmente sensiblement la souplesse de chaque structure de contrôle.

Répétition

La structure REPEAT/UNTIL

La structure de contrôle la plus simple, REPEAT/UNTIL, indique seulement à l'ordinateur de répéter l'exécution d'une instruction. Elle est illustrée dans la procédure de la Figure 3.24. On remarque que les mots REPEAT et UNTIL peuvent encadrer un nombre d'instructions quelconque. De plus, la syntaxe et les mots sont ceux du langage courant (REPEAT : REPETER, UNTIL : JUSQU'A). Chaque itération du processus est suivie d'un test dont le résultat spécifie si ce processus doit être répété. Par conséquent,

une structure REPEAT/UNTIL se termine toujours par un test. Il peut s'agir d'un test booléen (tel que UNTIL ToutFini = VRAI ou simplement UNTIL ToutFini) ou d'un test non booléen comme dans la Figure 3.24. Dans cet exemple, l'ordinateur évalue si l'expression "Etoiles = 5" est VRAI plutôt que de tester la valeur de Etoiles. La distinction intéresse principalement ceux qui écrivent des compilateurs, mais elle est mentionnée ici pour faire remarquer que cette structure, comme certaines autres, est contrôlée par les résultats d'un test booléen et non par un "compteur d'itérations".

On peut effectuer des tests autres qu'un test d'égalité :

```
UNTIL Etoiles > 5 ;
UNTIL Etoiles >= 5 ;
UNTIL Etoiles = Limite ;
UNTIL Etoiles > (Limite - 3) ;
```

Le test doit avoir un sens ; si on tape :

```
UNTIL Etoiles < Limite ;
```

et que Etoiles est déjà supérieur à Limite, le programme se déroule indéfiniment. Il ne s'agit évidemment pas d'un *bug* du Turbo mais d'une erreur de programmation.

```
PROCEDURE Repeat_Simple;
BEGIN
   Etoiles := 0;
   REPEAT
      WRITE ('* ');
      Etoiles := Etoiles + 1;
   UNTIL Etoiles = 5;
END;
```

Figure 3.24 : Programme illustrant une répétition simple.

La boucle WHILE/DO

Une autre méthode de réalisation de répétition simple apparaît Figure 3.25.

Cette structure emploie les mots réservés WHILE et DO. On

remarque que ces mots peuvent être *suivis* d'un nombre quelconque d'instructions. Comme précédemment, les mots et la syntaxe sont ceux du langage courant (WHILE : TANT QUE, DO : FAIRE). Chaque itération de la structure est précédée d'un test déclenchant ou non la répétition. Par conséquent, une structure WHILE/DO commence toujours par un test, qu'il soit booléen (tel que "WHILE ToutFini = FALSE ") ou non booléen comme dans la Figure 3.25.

Comparaison de structures de répétition

On peut se demander dans quelle situation employer la structure REPEAT/UNTIL plutôt que la structure WHILE/DO ; le programme de la Figure 3.26 combine des versions légèrement modifiées des procédures utilisées dans les Figures 3.24 et 3.25. Il montre comment les deux structures réagissent à la même entrée de données.

La Figure 3.27 montre l'écran de résultat généré par ce programme. La valeur 0 est entrée comme valeur de départ de chaque répétition. On remarque que chaque procédure est identifiée par un nom qui permet de connaître sa fonction. Dans cette série d'exemples, on demande l'affichage d'étoiles. Le nombre maximal d'étoiles est établi par la constante Limite qui a pour valeur 5.

```
PROCEDURE While_Simple;
BEGIN
   Etoiles := 0;
   WHILE Etoiles < 5 DO
   BEGIN
      WRITE ('* ');
      Etoiles := Etoiles + 1;
   END;
END;
```

Figure 3.25 : Programme utilisant une structure de contrôle WHILE/DO.

```
PROGRAM Demo_Structures_de_Controle;

CONST
   Limite = 5;

VAR
   Etoiles, Point_depart : INTEGER;

PROCEDURE While_Simple;
BEGIN
   WRITELN ('While simple');
   Etoiles := Point_depart;
   WHILE Etoiles < Limite DO
   BEGIN
      WRITE (' * ');
      Etoiles := Etoiles + 1;
   END;
   WRITELN
END;

PROCEDURE Repeat_Simple;
BEGIN
   Etoiles := Point_depart;
   WRITELN ('Repeat simple');
   REPEAT
      WRITE (' * ');
      Etoiles := Etoiles + 1;
   UNTIL Etoiles > Limite;
   WRITELN
END;

BEGIN
   WRITELN ('Nombre d étoiles pour débuter :');
   READLN (Point_depart);
   While_Simple;
   Repeat_Simple;
END.
```

Figure 3.26 : Programme comparant les structures REPEAT et WHILE.

```
Nombre d étoiles pour débuter :
0
While simple
 *  *  *  *  *
Repeat simple
 *  *  *  *  *  *
```

Figure 3.27 : Écran de résultats du programme de la Figure 3.26.

On remarque que la structure WHILE affiche 5 étoiles alors que la structure REPEAT en affiche 6. Le test doit être modifié si on veut en obtenir 5 ; dans ce cas, la procédure est celle de la Figure 3.28.

```
PROCEDURE Repeat_Simple;
BEGIN
   Etoiles := Point_depart;
   WRITELN ('Repeat simple');
   REPEAT
      WRITE (' * ');
      Etoiles := Etoiles + 1;
   UNTIL Etoiles > Limite - 1;         {Modification de la borne}
   WRITELN
END;
```

Figure 3.28 : Programme de la Figure 3.24 modifié pour générer le même résultat que la structure WHILE.

La Figure 3.29 montre le résultat obtenu avec le programme initial et avec une entrée différente.

```
Nombre d étoiles pour débuter :
4
While simple
 *
Repeat simple
 *  *
```

Figure 3.29 : Résultat du programme de la Figure 3.26 avec comme valeur de départ 4.

Jusqu'à présent, le programme s'est déroulé comme prévu. On examine maintenant la Figure 3.30 dans laquelle la valeur de départ et la limite ont toutes deux la valeur 5.

```
Nombre d étoiles pour débuter :
5
While simple

Repeat simple
 *
```

Figure 3.30 : Résultat du programme de la Figure 3.26 avec une valeur de départ égale à la limite.

La structure WHILE/DO n'affiche aucune étoile alors que la structure REPEAT/UNTIL en affiche une. On examine maintenant la Figure 3.31 dans laquelle le point de départ est supérieur à la limite.

```
Nombre d étoiles pour débuter :
20
While simple

Repeat simple
 *
```

Figure 3.31 : Résultat du programme de la Figure 3.26 avec une valeur de départ supérieure à la limite.

On obtient encore une étoile avec la boucle REPEAT/UNTIL. Comme précédemment, il ne s'agit pas d'un bug mais plutôt d'une qualité du Turbo. Celui-ci offre deux structures de contrôle de répétition simples ; l'une exécute toujours la boucle au moins une fois tandis que l'autre ne l'exécute que si les conditions du test sont satisfaites.

Cette caractéristique est fondamentale et sera largement illustrée au cours de cet ouvrage. La distinction entre les deux possibilités sera rapidement assimilée.

Pour mieux comprendre ces deux structures, on modifie la procédure REPEAT/UNTIL pour qu'elle ressemble à celle de la

Figure 3.32 puis on exécute le programme avec le jeu de données utilisé dans l'autre exemple.

```
PROCEDURE Repeat_Simple;
BEGIN
   Etoiles := Point_depart;
   WRITELN ('Repeat simple');
   REPEAT
      WRITE (' * ');
      Etoiles := Etoiles + 1;
   UNTIL Etoiles = Limite;
   WRITELN;
END;
```

Figure 3.32 : Programme illustrant l'oubli d'une limite pour la structure de contrôle de répétition.

Les boucles FOR/TO/DO

On remarque, dans les exemples précédents, qu'il fallait inclure une ligne dans l'instruction composée pour incrémenter le "compteur d'étoiles" :

Etoiles : =Etoiles + 1

La répétition avec incrémentation est tellement courante que le Turbo possède une structure qui effectue simultanément la répétition des instructions et l'incrémentation d'un compteur. Elle est semblable à la commande FOR/TO/STEP du BASIC. La Figure 3.33 montre notre exemple modifié pour être associé à une structure FOR/TO/DO.

On exécute le programme de la Figure 3.33 en utilisant les valeurs de départ 0, 1, 3, 5, 20 et – 12 pour se faire une idée du fonctionnement de la structure. On remarque que celle-ci ne teste pas de condition. Il suffit de lui indiquer les valeurs des limites pour qu'elle exécute la commande autant de fois que cela est spécifié.

```
PROGRAM Demo_Structures_de_Controle;

CONST
   Limite : INTEGER = 5;

VAR
   Etoiles, Point_depart : INTEGER;

PROCEDURE For_Do;
BEGIN
   WRITELN ('Boucle For Do simple');
   Etoiles := Point_depart;
   FOR Etoiles := Point_depart TO Limite DO
   BEGIN
      WRITE (' * ');
      Etoiles := Etoiles + 1;
   END;
   WRITELN;
END;


BEGIN
   WRITELN ('Nombre d étoiles pour débuter :');
   READLN (Point_depart);
   For_Do
END.
```

Figure 3.33 : Programme de démonstration des boucles FOR/TO/DO.

Dans la structure FOR/TO/DO, DOWNTO peut remplacer TO. De plus, on peut employer n'importe quel type scalaire (excepté les nombres réels) pour contrôler la boucle. La Figure 3.34 illustre l'utilisation de caractères comme variables de contrôle.

On exécute le programme de la Figure 3.34 avec différents caractères de début et de fin ; la table des codes ASCII permet de comprendre la séquence affichée.

```
PROGRAM Caracteres_variables_de_Controle;

VAR
   Caractere, Prem_car, Der_car : CHAR;

PROCEDURE For_Do;
BEGIN
   FOR Caractere := Prem_car TO Der_car DO
   BEGIN
      WRITE (Caractere);
   END;
   WRITELN;
END;

BEGIN
   WRITELN ('Premier caractère de l alphabet :');
   READLN (Prem_car);
   WRITELN ('Dernier caractère de l alphabet :');
   READLN (Der_Car);
   For_Do;
END.
```

Figure 3.34 : Programme de démonstration de contrôle de boucle avec des caractères.

Chacune de ces structures de répétition a un rôle particulier et des restrictions. Il faut prendre garde à la structure choisie ; dans la plupart des cas, il s'agit simplement d'une préférence du programmeur, mais parfois le choix est déterminé par le contexte dans lequel le programme se déroule. Dans un cas comme celui-ci, la structure FOR est inutile puisqu'il faut donner le nombre d'itérations avant le début de la boucle. D'autre part, il n'y a aucun intérêt à avoir des tests et des instructions supplémentaires dans un programme pour lequel le nombre d'itérations est connu (par exemple, pour l'impression d'une ligne d'étoiles en haut d'un rapport).

Instructions conditionnelles

La répétition est utile mais la capacité de choisir ou non la répétition est encore plus puissante, surtout si plusieurs alternatives

sont possibles ; le Turbo propose un jeu de structures adaptées. Comme pour la répétition, ces structures ont une syntaxe proche de celle du langage courant.

Choix parmi deux directions : IF/THEN/ELSE

IF/THEN/ELSE est peut-être la structure de contrôle la plus intuitive du Turbo car elle reflète directement le type de décision prise dans la vie de tous les jours. Sa syntaxe est très simple :

```
IF Aujourd hui = Paie THEN WRITELN ('Buvons un coup !') ;
```

Dans ce cas, l'action est exécutée uniquement s'il s'agit d'un jour de paie. Sinon, le programme passe à l'instruction suivante. On peut ajouter une action alternative :

```
IF Aujourd_hui = Paie THEN WRITELN ('Buvons un coup !') ;
    ELSE WRITELN ('Peux-tu me preter de l'argent ?') ;
```

Il ne faut pas oublier que ces actions simples peuvent être remplacées par des instructions composées. Une partie de programme destinée à afficher une liste d'adresses apparaît Figure 3.35. Alors

```
PROCEDURE Affiche_Liste;
BEGIN
   IF Lignes_faites < 58 THEN          {tant que la page n'est pas remplie}
   BEGIN
      READ (Fichier_Lect, Membres);
      WRITELN (LST, Membres.Nom);
      WRITELN (LST, Membres.Rue);
      WRITELN (LSΤ, Membres.Ville, ' ', Membres.Code_Post);
      Lignes_Faites := Lignes_Faites + 1;
   END;
   ELSE                                {alternative}
   BEGIN
      Lignes_Faites := 7;
      WRITELN (LST, #12);
              {envoi d'un Form-Feed pour générer un en-tête au début}
      WRITELN (LST, '        Liste des membres de l equipe H.M.S Pinafore');
      WRITELN (LST, '                                        1er Avril 1986');
      WRITELN; WRITELN; WRITELN;
      Lignes_Faites := 4;
   END;
END;
```

Figure 3.35 : Partie de programme illustrant les instructions composées multilignes.

que certains de ces détails peuvent sembler encore un peu obscurs, le principe de l'utilisation d'instructions composées multilignes doit être clair.

Comme nous l'avons mentionné précédemment, des tests peuvent être imbriqués. Cette technique est particulièrement pratique pour filtrer des données non valides. Dans la Figure 3.36, le pro-

```
PROGRAM Demo_Imbrication;


VAR
   Caractere, Prem_car, Der_Car : CHAR;

PROCEDURE Lect_donnees;
BEGIN
   WRITELN;
   WRITELN ('Sous-programme de lecture de données');
   WRITELN;
   WRITELN ('Premier caractère de l alphabet :');
   READLN (Prem_Car);
   WRITELN ('Dernier caractère de l alphabet :');
   READLN (Der_Car);
   IF (Prem_Car >= 'a') AND (Prem_Car <= 'z') THEN
      IF (Der_Car <= 'z') AND (Der_Car >= Prem_Car) THEN
         WRITELN ('Traitement de l entrée valide !')
      ELSE Lect_donnees
   ELSE Lect_donnees
END;

PROCEDURE For_Do;
BEGIN
   FOR Caractere := Prem_Car TO Der_Car DO
      WRITE (Caractere);
   WRITELN;
END;

BEGIN
   Lect_Donnees;
   For_Do;
END.
```

Figure 3.36 : Programme illustrant les structures IF/THEN imbriquées.

gramme ne permet pas l'entrée de caractères autres que des lettres minuscules. Bien que ce programme utilise aussi certains tests composés, on doit comprendre ce qui se passe en le lisant. On remarque que la procédure Lect_données est capable de s'appeler elle-même indéfiniment tant qu'elle reçoit des données non autorisées. On peut exécuter le programme avec des données autorisées ou non pour voir comment il se comporte, puis modifier les tests (en remplaçant par exemple >= par >) pour observer son nouveau comportement.

Alternatives multiples : CASE

Lorsqu'une décision résulte d'un choix plus complexe, le Turbo propose la structure CASE/OF. Cette structure extrêmement puissante se prête à de nombreuses opérations dont la plus importante consiste à prendre une direction en fonction d'un choix effectué dans un menu. La Figure 3.37 montre un exemple caractéristique de cette commande. Lorsque le menu est affiché, l'utilisateur tape un nombre qui est lu comme une option ; le choix est fait parmi 8 programmes. Si un nombre non autorisé est entré, la structure CASE permet d'employer ELSE pour traiter toute autre valeur.

```
PROCEDURE Interprete_choix;              {Interprétation d'un menu de sélection}
BEGIN
   READLN (Option);
   CASE Option OF
      1: Accepte_Nouv_Noms;
      2: Liste_Alphabetique;
      3: Tri_Etiquettes_Code_postal;
      4: Affiche_Liste_Sur_Ecran;
      5: Modification_Une_Entree;
      6: Creation_Liste_Adresses;
      7: Sortie;
   ELSE Gestion_erreur;
   END;
END;
```

Figure 3.37 : Programme de démonstration d'une structure CASE.

La variable test (Option dans cet exemple) est une valeur scalaire non réelle quelconque. Les données booléennes entières et caractère sont toutes autorisées. Le nombre des différents choix proposés par l'instruction CASE est illimité.

Le programme de la Figure 3.38 montre une technique permettant de convertir des nombres réels en nombres entiers. On exécute le programme en entrant des nombres réels très proches des limites et on observe le résultat. Ce programme illustre également la manière dont le Turbo arrondit et évalue les nombres réels.

```
PROGRAM Demo_Case;

VAR
   A : REAL;
   B : INTEGER;

PROCEDURE Faire;
BEGIN
   WRITELN ('Taper un nombre :');
   READLN (A);
   IF (A <= 2) THEN B := 2;
   IF (A > 2) AND (A < 4) THEN B := 4;
   IF (A >= 4) AND (A <= 6) THEN B := 6;
   IF (A > 6) THEN B := 7;
   CASE B OF
      2: WRITELN ('2 ou moins');
      4: WRITELN ('2 à 4');
      6: WRITELN ('4 à 6');
   ELSE
      WRITELN ('supérieur à 6');
   END;
   Faire;
END;

BEGIN
   Faire;
END.
```

Figure 3.38 : Programme de démonstration de structure CASE avec une entrée de nombres réels.

La structure GOTO/LABEL

Le Turbo propose également une structure *inconditionnelle* pour les situations de programmation dans lesquelles on veut toujours détourner le déroulement du programme d'une exécution séquentielle. Cette structure, GOTO/LABEL, ne teste pas une condition ou un compteur mais exécute la ligne de programme spécifiée par l'étiquette indiquée. Le Turbo emploie assez peu l'instruction GOTO et restreint sévèrement la distance qu'elle peut couvrir. Cependant, elle peut représenter un moyen d'éviter une boucle ou de mettre fin à un programme.

La plupart des programmeurs Turbo mettent un point d'honneur à ne jamais employer cette commande. En effet, la gamme de structures de contrôle proposée par le Turbo doit autoriser toute programmation sans branchement inconditionnel. Néanmoins, le rôle de la programmation est de résoudre des problèmes aussi simplement que possible ; par conséquent, si l'instruction GOTO est préférable à l'instruction WHILE dans certaines situations, il faut l'employer. Les étiquettes doivent être déclarées avant leur emploi. Cependant, le format de déclaration est différent de celui des autres variables. Une étiquette n'a pas de type et ne comprend pas les signes suivants : égal (=), deux points (:) et opérateurs d'assignation. Le programme ci-dessous montre comment les étiquettes sont déclarées et employées :

```
PROCEDURE Montre_Etiquette
LABEL
        Exemple_Etiquette ;
BEGIN
        WRITELN ('Bonjour') ;
        Exemple_Etiquette :
                WRITELN ('Ceci est la premiere ligne à executer
                lorsque le Turbo rencontre l'instruction GOTO
                Exemple_Etiquette') ;
.
.
.
        GOTO Exemple_Etiquette ;
END ;
```

Il ne faut pas oublier que l'instruction GOTO travaille uniquement à l'intérieur d'un bloc. On ne peut pas entrer dans une procédure ou dans une fonction (ni en sortir) à l'aide de cette instruction.

On voit dans les programmes plus sophistiqués de ce livre que pour passer d'une procédure à une autre il suffit de l'appeler par son nom. L'emploi le plus courant de l'instruction GOTO dans des langages tels que le BASIC (déroutement du programme vers un sous-programme) correspond en Turbo à l'appel d'une procédure par son nom. Lorsque ces noms sont significatifs, cette technique rend le déroulement très clair pour l'utilisateur.

Étant donné qu'il est impossible de sortir d'un module avec une instruction GOTO, les définitions d'étiquettes font en principe partie de la section de déclaration de procédures.

Le Turbo offre également deux commandes standard, HALT et EXIT, permettant de sortir d'une procédure ou d'un programme. Quel que soit le déroulement en cours, la commande HALT met fin au programme. Si le fichier a été compilé, le programme revient au signal du système d'exploitation ; si l'opération se déroule dans l'environnement Turbo, il revient au signal Turbo sans en sortir.

La commande EXIT a un caractère conditionnel ; exécutée dans le corps principal du programme, elle opère exactement comme la commande HALT. Exécutée à l'intérieur d'une procédure ou d'une fonction, elle permet de sortir de ce module et le déroulement continue à partir de la ligne suivant l'appel de la procédure. Par conséquent, EXIT sert de soupape de sécurité en cas d'erreur.

PROCÉDURES ET FONCTIONS : OUTILS DU TURBO

On a remarqué que les exemples de programme de ce livre appellent des procédures distinctes et non de longs modules composés de lignes de code séquentielles. On a maintenant une bonne vue d'ensemble de l'aspect et de l'organisation générale d'un programme en Turbo Pascal. L'idée qu'un petit module codé placé à la fin du programme appelle toutes les procédures précédemment créées semble naturelle. Il est temps maintenant d'explorer des concepts relatifs aux procédures et aux fonctions plus approfondis, et de voir comment il est possible de les employer. Nous allons voir que l'organisation d'un programme Turbo avec des procédures est une aide puissante pour l'écriture de programmes dont la mise au point est aisée.

Nous allons commencer par différencier les procédures et les fonctions. Celles-ci sont similaires et ressemblent aux programmes et aux sous-programmes des autres langages de programmation. En programmation Turbo, il est toujours possible de remplacer une

instruction simple par une instruction composée. De même, on peut toujours employer une procédure à la place d'une instruction. Si on regarde les exemples de programmes abordés précédemment, on s'aperçoit qu'à chaque emploi de procédure il aurait été possible de substituer une instruction simple ou composée. Les procédures autorisent donc l'accès à des blocs de code plus facilement, plus clairement et plus naturellement qu'en insérant des lignes de code à différents points du programme. Pour une compréhension optimale, il est conseillé de ne pas écrire de modules dont la taille dépasse celle d'un écran. Par conséquent, les appels à des procédures distinctes pour exécuter des tâches spécifiques (accès aux informations, conversion de caractères, impression d'un menu, choix d'une option, etc.) clarifient l'ensemble du programme. La netteté et l'intelligibilité ne sont pas seulement utiles lorsque l'on revient à un programme écrit longtemps auparavant. La répartition des tâches d'un long programme en modules codés et mis au point séparément est l'une des clés du développement rapide de programmes dépourvus de bugs.

Les fonctions renvoient toujours une valeur au programme d'appel. Comme on utilise un identificateur de variables dans un programme lorsqu'il est nécessaire de manipuler la valeur qu'elle représente, on emploie des fonctions pour manipuler ces valeurs. Par exemple, il existe des fonctions standard qui donnent le carré d'un nombre ou sa racine carrée. Les instructions suivantes sont équivalentes :

```
Aire = Cote * Cote
Aire = SQR (Cote) {Utiliser la fonction SQR}
```

Si X est un nombre égal à 7.525 et JoursDansLaSemaine un entier égal à 7, les instructions suivantes sont équivalentes :

```
PartieEntiere = Trunc(X) ;
PartieEntiere = Trunc(7.525) ;
PartieEntiere = 7 ;
PartieEntiere = 4 + 3 ;
PartieEntiere = JoursDansLaSemaine
```

Les fonctions définies par l'utilisateur servent à exécuter des opérations qui ne sont pas disponibles dans la gamme de procédures et de fonctions standard du Turbo. Il n'existe pas de commande Turbo pour calculer la puissance 3 d'un nombre ou

pour représenter le solde d'un compte après investissement d'un certain montant avec un taux d'intérêt donné pour X années. Dans le second cas, si un programme doit calculer de telles valeurs *plus d'une fois*, il faut employer une fonction qui prendra une valeur initiale et un temps donné pour générer puis renvoyer au programme d'appel le solde final.

Les sociétés de mise au point d'applications utilisent largement ce concept en constituant des bibliothèques de fonctions standard conçues spécifiquement pour le type de travail de la société et disponibles pour l'ensemble de ses programmeurs. Un bureau de gestion pourrait disposer d'une bibliothèque de tables d'amortissement. Il est donc aisé d'adapter le Turbo à des applications spécifiques.

Cependant, pour la programmation courante, le Turbo offre entre 40 et 60 fonctions prédéfinies (selon la version, le système d'exploitation et l'ordinateur employés) qui couvrent un nombre d'applications très important. La Figure 3.39 montre une table des fonctions prédéfinies pour la version 3.0 du Turbo sur IBM PC.

PROCÉDURES CRÉÉES PAR L'UTILISATEUR

Les procédures sont des sous-programmes qui rendent un long programme plus clair, plus lisible et plus facile à tester et à mettre au point. Elles peuvent être longues ou courtes mais commencent toutes par le mot réservé PROCEDURE et se terminent toutes par le mot réservé END ; (le point-virgule est *obligatoire*). Toutes les instructions placées entre ces mots réservés caractérisent *la portée* de la procédure. Les déclarations faites au début du programme sont effectives à l'intérieur de ses différentes procédures. Si l'on revient aux exemples de programmes précédents, on voit que les variables calculées ou entrées dans une procédure peuvent être ultérieurement manipulées dans les autres. Quand elle est entrée en dehors d'une procédure mais lue et traitée par une autre, cette valeur est appelée *paramètre* et le processus de transfert de la valeur *passage de paramètre*. Dans tous les exemples de programmes étudiés jusqu'à présent, le passage de paramètres était réalisé simplement en utilisant des variables communes à l'ensemble du programme ; cette manière de procéder est identique dans d'autres langages tels que le BASIC.

Les abréviations de cette liste sont utilisées pour représenter les arguments des fonctions suivantes :

B	Booléen
C	Caractère
E	Valeur entière ou valeur réelle
F	Variable fichier
FONC	Numéro de fonction du système d'exploitation (BIOS ou BDOS)
I	Entier
POS	Position dans une chaîne
PTR	Variable pointeur
R	Réel
SC	Valeur scalaire quelconque
ST	Chaîne de caractères
SUB	Sous-chaîne de caractères
V	Variable quelconque
X	Variable particulière décrite dans le descriptif associé.

Sauf précision, le résultat d'une procédure ou d'une fonction est du même type que celui de l'argument associé ; d'autre part, certaines fonctions et procédures sont liées à plusieurs descriptifs.

Figure 3.39 : Fonctions prédéfinies du Turbo Pascal.

Fonctions et procédures arithmétiques

ABS (E)	Retourne la valeur absolue de E.
EXP (E)	Retourne l'exponentielle de E.
FRAC (E)	Retourne la partie fractionnaire de E ; le résultat est toujours un réel.
INT (E)	Retourne la partie entière de E ; le résultat est toujours un réel.
LN (E)	Retourne le logarithme naturel (népérien) de E ; le résultat est toujours un réel.
RANDOM	Retourne un nombre aléatoire supérieur ou égal à zéro et inférieur à un ;le résultat est toujours un réel.
RANDOM (I)	Retourne un nombre aléatoire supérieur ou égal à zéro et inférieur à I ; le résultat est toujours un entier.
ROUND (R)	Retourne le nombre R arrondi à l'entier le plus proche ; le résultat est toujours un entier.
SQR (E)	Retourne le carré du nombre E (E * E).
SQRT (E)	Retourne la racine carrée de E ; le résultat est toujours un réel.
TRUNC (R)	Retourne le plus grand entier inférieur ou égal à R ; le résultat est toujours un entier.

Fonctions de conversion, traduction et transfert

CHR (I)	Retourne le caractère ASCII correspondant à la valeur ordinale I.

Figure 3.39 (suite)

HI (I)	Retourne dans l'octet de poids faible l'octet de poids fort de la valeur I. L'octet de poids fort du résultat contient 0 ; le résultat est toujours un entier.
LO (I)	Retourne dans l'octet de poids faible l'octet de poids faible de la valeur I. L'octet de poids fort du résultat contient 0 ; le résultat est toujours un entier.
ORD (SC)	Retourne la valeur ordinale de SC dans l'ensemble défini par le type de SC ; le résultat est toujours un entier et SC ne peut pas être un réel. Cette fonction opère de manière très différente avec les pointeurs (voir plus loin).
ROUND (R)	Retourne la valeur R arrondie à l'entier le plus proche ; le résultat est toujours un entier.
STR (E, ST)	Convertit la valeur numérique E en une chaîne et la stocke dans dans ST. E doit être un paramètre d'écriture de type entier ou de type réel.
SWAP (I)	Retourne une valeur dans laquelle l'octet de poids fort correspond à l'octet de poids faible de I et dans laquelle l'octet de poids faible correspond à l'octet de poids fort de I ; le résultat est toujours un entier.
TRUNC (R)	Retourne l'entier le plus grand inférieur ou égal à R ; le résultat est toujours un entier.
UPCASE (C)	Retourne la majuscule équivalente au caractère C.
VAL (ST, E, I)	Convertit l'expression de chaîne ST en une valeur réelle ou une valeur entière et stocke le résultat dans E. Si une erreur apparaît, I correspond à la position du caractère ayant causé celle-ci.

Figure 3.39 (suite)

ADDR (X)	Retourne l'adresse absolue en mémoire du premier octet de la variable, tableau, enregistrement, procédure ou fonction. L'adresse d'un élément particulier de tableau ou d'enregistrement peut aussi être obtenu en utilisant l'indexe approprié. Le résultat est toujours un entier.
BDOS (FONC, I)	Procédure permettant d'appeler la fonction BDOS de CP/M de numéro FONC ; I est un entier facultatif qui est chargé dans le registre DE. Un appel à l'adresse 5 exécute ensuite la fonction BDOS.
BDOS	Fonction retournant la valeur entière chargé dans le registre A par le BDOS de CP/M à la suite de l'exécution d'une procédure BDOS ou BDOSHL.
BDOSHL (FONC, I)	Procédure permettant d'appeler la fonction BDOS de CP/M de numéro FONC ; I est un entier facultatif qui est chargé dans le registre HL. Un appel à l'adresse 5 exécute ensuite la fonction BDOS.
BIOS (FONC, I)	Procédure permettant d'appeler la fonction BIOS de CP/M de numéro FONC ; I est un entier facultatif qui est chargé dans le registre BC.
BIOS	Fonction retournant la valeur entière chargé dans le registre A par le BIOS de CP/M à la suite de l'exécution d'une procédure BIOS ou BIOSHL.
BIOSHL (FONC, I)	Procédure permettant d'appeler la fonction BIOS de CP/M de numéro FONC ; I est un entier facultatif qui est chargé dans le registre HL.
OVRDRIVE (I)	Procédure permettant de changer le numéro du lecteur de disque en cours pour la lecture de fichiers de recouvrement (OVERLAY) ; la valeur I spécifie le numéro du lecteur (0 = lecteur en cours, 1 = lecteur A, 2 = lecteur B, etc ...)

Figure 3.39 (suite)

Fonctions de gestion de fichiers

ASSIGN (F, ST)	Assigne le nom de fichier stocké dans la variable ST à la variable de fichier F.
CLOSE (F)	Ferme le fichier F et réalise la mise à jour dans le répertoire associé.
EOF (F)	Retourne une valeur booléenne TRUE (vrai) si le pointeur de fichier est placé au-delà du dernier élément du fichier disque F.
EOLN (F)	Retourne une valeur booléenne TRUE (vrai) si le pointeur de fichier est positionné sur le caractère "retour chariot" dans le fichier texte F. Dans un fichier texte, si EOF (F) est vrai, alors EOLN (F) est aussi vrai.
ERASE (F)	Supprime le fichier disque F.
FILEPOS (F)	Retourne la position en cours du pointeur de fichier de F ; le résultat est toujours un entier. Cette instruction ne peut pas être associée à des fichiers texte.
FILESIZE (F)	Retourne la taille du fichier disque F exprimée en nombre d'éléments de ce fichier (pour un fichier d'enregistrements, on obtient un nombre d'enregistrements) ; le résultat est toujours un entier. Cette instruction ne peut pas être associées à des fichier texte.
FLUSH (F)	Vide la mémoire tampon interne du fichier disque F ; cette opération est faite automatiquement et cette fonction est rarement utilisée par un programme. Cette instruction ne peut pas être associée à des fichiers texte.

Figure 3.39 (suite)

RENAME (F, ST)	Renomme le fichier disque F avec le nom de fichier stocké dans la variable ST.
REWRITE (F)	Prépare le nouveau fichier disque F pour un traitement ; s'il existe déjà, le fichier F sera effacé puis créé à nouveau. Le pointeur de fichier est positionné au début.
RESET (F)	Prépare le fichier disque F (déjà créé) pour un traitement ; si le fichier n'existe pas, une erreur d'entrée/sortie est générée. Le pointeur de fichier est positionné au début.
SEEK (F, I)	Déplace le pointeur de fichier sur le Ième composant du fichier F. Cette instruction ne peut pas être associée à des fichiers texte.
SEEKEOF (F)	Retourne la valeur booléenne TRUE (vrai) si le pointeur de fichier est positionné au-delà du dernier composant du fichier F. A l'inverse de EOF, SEEKEOF écarte les blancs, les tabulations et les marques de fin de ligne avant de tester la fin du fichier.
SEEKEOLN (F)	Retourne la valeur booléenne TRUE (vrai) si le pointeur de fichier est positionné dans le fichier texte F sur un caractère "retour chariot". A l'inverse de EOLN, SEEKEOLN écarte les blancs et les tabulations avant de tester le caractère "retour chariot". Dans un fichier texte, si EOF (F) est vrai, alors SEEKEOLN (F) est aussi vrai.

Fonctions relatives au tas et aux pointeur

DISPOSE (PTR)	Récupère dans le tas la mémoire utilisée par la variable pointeur PTR.

Figure 3.39 (suite)

FREEMEM (PTR, I)	Récupère dans le tas I octets de mémoire pour la variable pointeur PRT ; le nombre d'octets doit être exactement le même que celui qui a été préalablement affecté à la variable PTR par la fonction GETMEM.
GETMEM (PTR, I)	Alloue I octets de mémoire dans le tas pour la variable pointeur PTR.
MARK (PTR)	Procédure : assigne le pointeur de tas à la variable PTR.
MAXAVAIL	Retourne la taille du plus grand espace disponible dans le tas. Le résultat est toujours un entier mais permet plusieurs interprétations s'il est supérieur à MAXINT.
MEMAVAIL	Retourne le nombre de paragraphes (16 octets) libres dans le tas ; le résultat est toujours un entier.
NEW (PTR)	Crée le pointeur de variable PTR.
ORD (PTR)	Retourne l'adresse contenue dans le pointeur PTR ; le résultat est toujours un entier. Le résultat de cette fonction est différent si l'argument est un scalaire.
PTR (I)	Convertit la valeur entière I en un pointeur.
RELEASE (PTR)	Supprime toutes les variables dynamiques au-dessus de l'adresse PTR (PTR est préalablement initialisée par la procédure MARK).

Fonctions d'entrée/sortie

BLOCKREAD (F, V, I)	Lit, à partir du fichier disque sans type F, I enregistrements de 128 octets et les place dans la variable V.

Figure 3.39 (suite)

BLOCKWRITE (F, V, I) Ecrit, à partir de la variable V, I enregistrements de 128 octets dans le fichier disque sans type F.

IORESULT — Retourne une valeur correspondant au code d'erreur E/S si une erreur de ce type a eu lieu. L'appel de la fonction IORESULT après une erreur d'entrée/sortie remet à zéro la condition d'erreur et permet au programme d'effectuer de nouvelles opérations d'entrée/sortie.

La plupart des entrées/sorties sont effectuées avec les quatre procédures suivantes; les parties relatives aux entrées/sorties et aux fichiers illustrent les paramètres et les options disponibles. Ces parties décrivent aussi l'usage de ces procédures avec tous types de fichiers, périphériques logiques, fichiers standards etc...

READ (F, V..V)

READLN (F, V..V)

WRITE (F, V..V)

WRITELN (F, V..V)

Fonctions scalaires

ODD (I) — Retourne la valeur booléenne TRUE (vrai) si I est impair.

PRED (SC) — Renvoie le prédecesseur de SC s'il existe.

SUCC (SC) — Renvoie le successeur de SC s'il existe.

Figure 3.39 (suite)

Procédures et fonctions relatives à l'écran et au clavier

CLREOL	Efface tous les caractères depuis la position du curseur et jusqu'à la fin de la ligne sans déplacer le curseur.
CLRSCR	Efface la totalité de l'écran et place le curseur dans le coin supérieur gauche.
CRTINIT	Envoie à l'écran la chaîne d'initialisation de terminal (utilisée très rarement).
CRTEXIT	Envoie à l'écran la chaîne de réinitialisation de terminal (utilisée très rarement).
DELLINE	Supprime la ligne sur laquelle le curseur est placé et déplace les lignes en-desous d'une ligne vers le haut.
FILLCHAR (V,I,VAL)	Remplit I octets de la mémoire à partir du premier octet occupé par V avec la valeur VAL (de type caractère ou de type octet).
GOTOXY (X, Y)	Place le curseur au point de coordonnées X, Y ; X est une position horizontale et Y est une position verticale. Ce sont des valeurs entières mesurées à partir du coin supérieur gauche de l'écran.
HIGHVIDEO	Envoie à l'écran l'attribut vidéo HIGH comme défini dans la procédure d'installation de terminal.
INSLINE	Insère une ligne vide à la position du curseur.
KEYPRESSED	Retourne la valeur booléenne TRUE (vrai) si une touche du clavier a été pressée.
LOWVIDEO	Envoie à l'écran l'attribut vidéo LOW comme défini dans la procédure d'installation de terminal.

Figure 3.39 (suite)

MOVE (V1, V2, I) Effectue une copie de la zone mémoire de I octets, commençant à partir du premier octet de la variable V1 ; cette copie est réalisée à partir du premier octet de la variable V2. Cette instruction était principalement utilisée pour des déplacements d'images graphiques à grande vitesse.

NORMVIDEO Envoie à l'écran l'attribut vidéo NORM comme défini dans la procédure d'installation de terminal.

Fonctions de manipulation de chaînes

CONCAT (ST..ST) Concatène un nombre quelconque d'expressions de chaîne séparées par des virgules ; le résultat est toujours une chaîne. La même fonction peut être réalisée en utilisant le signe plus comme opérateur de concaténation.

COPY (ST, POS, I) Retourne une sous-chaîne, de la chaîne ST, de I caractères de long et commençant à partir de la position POS ; le résultat est toujours une chaîne.

DELETE (ST,POS,I) Supprime une sous-chaîne, dans la chaîne ST, de I caractères de long et commençant à partir de la position POS ; le résultat est toujours une chaîne.

INSERT (SU,ST,POS) Insère la chaîne SU dans la chaîne ST à partir de la position POS.

LENGTH (ST) Retourne le nombre de caractère de la chaîne ST ; le résultat est toujours un entier.

POS (ST1, ST2) Retourne la position du début de la chaîne ST1 dans la chaîne ST2 ; le résultat est toujours un entier.

STR (E, ST) Convertit la valeur numérique E en une chaîne ST ; E doit être un paramètre d'écriture entier ou réel.

Figure 3.39 (suite)

VAL (ST, E, I)	Convertit l'expression de chaîne ST en une valeur E entière ou réelle. Si une erreur apparaît, I contient la position du caractère qui a généré l'erreur.

Fonctions trigonométriques

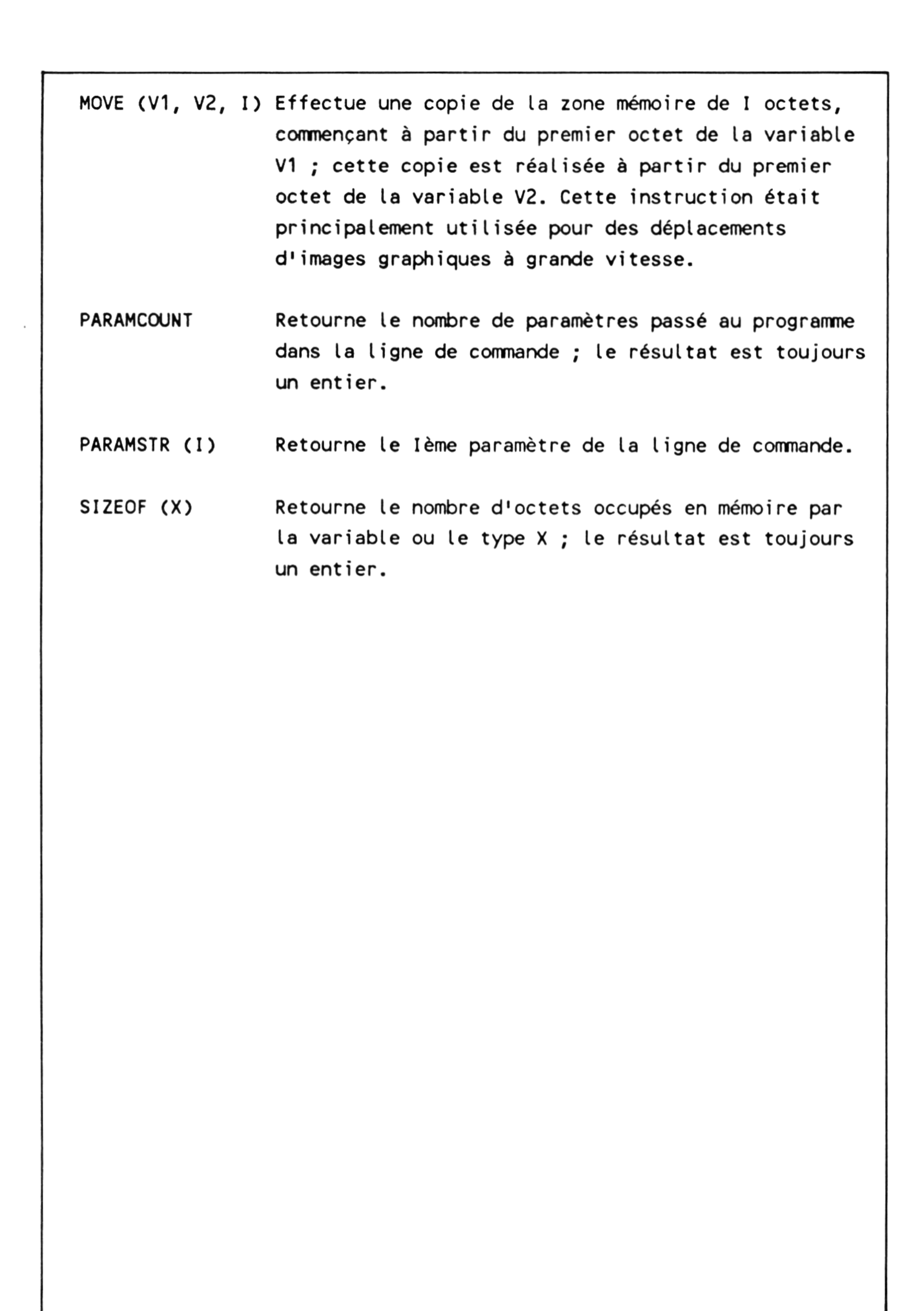
Tous les angles sont exprimés en radians.

ARCTAN (E)	Arctangente : retourne l'angle dont la tangente est E.
COS (E)	Cosinus : retourne le cosinus de l'angle E.
SIN (E)	Sinus : retourne le sinus de l'angle E.

Fonctions diverses

DELAY (I)	Exécute une boucle d'attente d'environ I millisecondes.
EXIT	Sort du bloc en cours. Si la fonction est appelée à l'intérieur d'une procédure ou d'une fonction, elle revient à la structure d'appel ; si elle est appelée dans le corps principal du programme, celui-ci est interrompu.
FILLCHAR (V, I, VAL)	Remplit I octets de la mémoire à partir du premier octet occupé par V avec la valeur VAL (de type caractère ou de type octet).
HALT	Arrête l'exécution du programme et revient au système d'exploitation.

Figure 3.39 (suite)



MOVE (V1, V2, I)	Effectue une copie de la zone mémoire de I octets, commençant à partir du premier octet de la variable V1 ; cette copie est réalisée à partir du premier octet de la variable V2. Cette instruction était principalement utilisée pour des déplacements d'images graphiques à grande vitesse.
PARAMCOUNT	Retourne le nombre de paramètres passé au programme dans la ligne de commande ; le résultat est toujours un entier.
PARAMSTR (I)	Retourne le Ième paramètre de la ligne de commande.
SIZEOF (X)	Retourne le nombre d'octets occupés en mémoire par la variable ou le type X ; le résultat est toujours un entier.

Figure 3.39 (suite)

Autre note historique

Il ne faut pas se laisser intimider par les termes *paramètre* et *passage de paramètres*. Comme pour les *structures de contrôle* abordées précédemment, ces termes évoquent le temps révolu où les programmeurs émergeaient épuisés d'une nuit de travail passée à jongler avec le contenu des registres et des piles pour tenter désespérément de passer tous leurs paramètres. Le commun des mortels était supposé vouer une crainte révérencielle à tout individu capable d'appréhender un concept aussi obscur, obtus et complexe.

Ces pionniers de la programmation devaient garder une trace des valeurs chargées dans des registres individuels tout en protégeant les données qui n'avaient pas à être manipulées. Le passage de paramètres demandait une compréhension approfondie du travail interne de l'ordinateur, en particulier en ce qui concerne les piles et registres. Actuellement, le passage de paramètres en Turbo consiste uniquement à savoir quel type d'information mettre dans un programme pour obtenir le résultat désiré.

Nous allons revenir aux procédures et acquérir une certaine pratique du passage de paramètres. Il ne faut pas oublier que le Turbo ne possède pas de commandes pour exécuter des tabulations horizontales ; il nous faut pour cela créer notre propre commande. La procédure de la Figure 3.40 génère une tabulation simple. On entre une chaîne et une taille de champ puis la procédure centre cette chaîne à l'intérieur du champ.

On tape le programme puis on l'exécute. On remarque que les identificateurs Phrase et Champ sont utilisés dans le corps principal du programme ; on leur assigne des valeurs puis on appelle la procédure Tab qui opère sur les mêmes variables. Phrase et Champ sont les valeurs nécessaires à l'obtention de résultats, c'est-à-dire à la disposition de la phrase dans la page.

La procédure de la Figure 3.40 fonctionne mais peut être fastidieuse dans le cas de nombreuses tabulations. Pour chaque phrase à centrer, il faut une instruction qui assigne une valeur à la variable Phrase, une autre qui assigne une valeur à la variable Champ et une troisième qui appelle la procédure Tab. Cette procédure est acceptable pour une opération unique mais peu élégante comme outil de tabulation général. On peut exécuter la même tâche en une seule instruction qui appelle la procédure Tab, indique la phrase à centrer ainsi que la taille du champ. De plus, il serait préférable de pouvoir choisir si le texte doit être centré, aligné à

droite ou à gauche.

```
PROGRAM Demo_tabulation;

VAR
   EspacesG, EspacesD, Phrase : STRING[80];
   A, I, Champ, EspacesCoteG, EspacesCoteD, EspacesSurLigne : INTEGER;

PROCEDURE Tab;                 {tabulation horizontale avec texte centré}
BEGIN
   EspacesG := '';
   EspacesD := '';
   EspacesSurLigne := Champ - LENGTH (Phrase);
   EspacesCoteG := EspacesSurLigne DIV 2;
   EspacesCoteD := (EspacesSurLigne DIV 2) + (EspacesSurLigne MOD 2);
   FOR I :=  1 TO EspacesCoteG DO EspacesG := EspacesG + ' ';
   FOR I :=  1 TO EspacesCoteD DO EspacesD := EspacesD + ' ';
   WRITE (LST, EspacesG, Phrase, EspacesD);
END;

BEGIN
   FOR A := 1 TO 3 DO                {test sur 3 essais}
   BEGIN
      WRITELN ('Tapez une phrase :');
      READLN (Phrase);
      WRITELN ('Largeur d une colonne :');
      READLN (Champ);
      Tab;
      WRITELN (LST);
   END;
END.
```

Figure 3.40 : Procédure de tabulation simple créée par l'utilisateur.

Le programme de la Figure 3.41 emploie une procédure Tab qui réalise toutes ces opérations. Dans tous les programmes de tabulation de cet ouvrage, la fonction génère des tirets à la place des espaces pour que le positionnement du texte soit plus clair. Lorsque l'opération de cette fonction est assimilée, on peut substituer un espace entre guillemets (" ") à chaque tiret entre guillemets (" - ") employé par le programme.

```
PROGRAM Demo_Fonctions;    {tabulation horizontale avec passage de paramètres}
TYPE
   Mots = STRING[80];
VAR
   EntreePhrase : Mots;
   A, Taille    : INTEGER;
   Position     : CHAR;
PROCEDURE Tab (Phrase : Mots; Champ : INTEGER; Pos : CHAR);
VAR
   EspacesG, EspacesD : STRING[80];
   I, EspacesCoteG, EspacesCoteD, EspacesSurLigne : INTEGER;
BEGIN
   EspacesG := '';
   EspacesD := '';
   EspacesSurLigne := Champ - LENGTH (Phrase);
   IF (Pos = 'C') OR (Pos = 'c') THEN  {centré}
   BEGIN
      EspacesCoteG := EspacesSurLigne DIV 2;
      EspacesCoteD := (EspacesSurLigne DIV 2) + (EspacesSurLigne MOD 2);
      FOR I :=  1 TO EspacesCoteG DO EspacesG := EspacesG + '_';
      FOR I :=  1 TO EspacesCoteD DO EspacesD := EspacesD + '_';
   END;
   IF (Pos = 'D') OR (Pos = 'd') THEN  {cadré à droite}
   BEGIN
      EspacesCoteG := EspacesSurLigne;
      FOR I :=  1 TO EspacesCoteG DO EspacesG := EspacesG + '_';
   END;

   IF (Pos = 'G') OR (Pos = 'g') THEN  {cadré à gauche}
   BEGIN
      EspacesCoteD := EspacesSurLigne;
      FOR I :=  1 TO EspacesCoteD DO EspacesD := EspacesD + '_';
   END;

   WRITE (LST, EspacesG, Phrase, EspacesD);
END;

BEGIN
   FOR A := 1 TO 3 DO
   BEGIN
      WRITELN ('Tapez une phrase :');
      READLN (EntreePhrase);
      WRITELN ('Largeur d une colonne :');
      READLN (Taille);
      WRITELN ('Gauche, Droite ou Centrée ?');
      READLN (Position);
      Tab (EntreePhrase, Taille, Position); {passage des paramètres}
      WRITELN (LST);
   END;
END.
```

Figure 3.41 : Programme de la Figure 3.40 modifié pour accepter le passage de paramètres.

On remarque sur la Figure 3.41 que la procédure Tab est appelée avec l'instruction unique :

Tab (EntreePhrase, Taille, Position)

Par exemple, la fonction Tab avec ces valeurs :

Tab (Salutations, 80, C)

centre le mot "Bonjour " (valeur assignée à la variable de chaîne Salutations) sur 80 colonnes. La version :

Tab (Nom, 40, G)

affiche le contenu de la variable Nom dans un champ de 40 caractères. Des espaces supplémentaires apparaissent à droite du nom.
Dans un programme d'application réel, cette procédure serait utilisée de la manière suivante :

```
{Le programme accede d'abord à un fichier de donnees pour
obtenir les informations necessaires.}
Tab (TotalMensuel, 40, C) ;
WRITELN (LST) ;
{Boucle lisant des enregistrements de clients successifs.}
Tab (NomClients, 25, G) ;
Tab (SoldeClients, 10, D) ;
WRITELN (LST) ;
{Le programme continue.}
```

Cette partie de programme génère un résultat du type de celui qui est présenté par la Figure 3.42.

```
____________________12 246____________________
Ale Amot______________________________245.47
Marcel Durand_________________________445.00
Gilbert Lemercier_______________________2.25
Gaetan Pis__________________________1 378.03
```

Figure 3.42 : Exemple de résultat imprimé après déroulement du programme de la Figure 3.41.

Il est évident que la seconde approche est plus élégante que la première. Elle évite la confusion liée à l'assignation des variables employées dans le corps du programme à celles de la procédure Tab. Par exemple, dans le programme précédent, il n'est pas nécessaire d'assigner l'identificateur NomClients à l'identificateur Phrase ; cette tâche est automatique lors de l'appel de la procédure.

On remarque, Figure 3.41, que la procédure Tab contient une section de déclaration. Les variables et les identificateurs déclarés à l'intérieur d'une procédure sont uniquement définis pour cette procédure. Si on essaie d'employer dans un autre endroit du programme les variables EspacesG, EspacesD, EspacesCôtéG, EspacesCôtéD et EspaceSurLigne, le Turbo affiche rapidement les messages *Unknown identifier* (identificateur inconnu) ou *Syntax error* (erreur de syntaxe).

Que gagne-t-on à employer ces variables *locales* ? Tout d'abord, ce processus permet de placer un groupe de procédures et de fonctions courantes dans un fichier bibliothèque indépendant. On peut combiner (ou inclure) ce fichier à un autre programme au moment de la compilation sans s'occuper d'extraire les informations de déclaration des procédures de la bibliothèque et en les plaçant au début du programme principal. La déclaration interne (locale) de chacune des procédures est tout à fait suffisante. De cette façon, le plus important se trouve à l'intérieur du programme principal sans qu'il y ait de restrictions relatives aux sous-programmes de la bibliothèque.

Il y a un autre avantage à employer des déclarations locales : lorsqu'une procédure a été mise au point et qu'elle fonctionne correctement, on peut l'utiliser simplement, comme s'il s'agissait d'une fonction prédéfinie. Il suffit de se souvenir de la syntaxe permettant d'y accéder sans s'occuper de ce qui se passe à l'intérieur de la procédure ; une boîte noire a ainsi été créée.

En fait, de nombreuses commandes employées jusqu'à présent (comme WRITELN et READLN) sont simplement des procédures prédéfinies auxquelles on passe des paramètres et qui exécutent un certain travail. Ce sont effectivement des boîtes noires utilisées dans chaque programme.

Les procédures et les fonctions ne se contentent pas d'accepter des entrées mais génèrent fréquemment des résultats. La Figure 3.43 montre le programme précédent modifié pour procéder à une mise en page sans effectuer l'affichage ; le programme d'appel s'en charge. Cette modification peut servir à introduire dans le programme d'appel une option destinée à diriger des informations

```
PROGRAM Demo_Fonctions;     {tabulation horizontale avec passage de paramètres}
TYPE
   Mots = STRING[80];
VAR
   EntreePhrase : Mots;
   A, Taille    : INTEGER;
   Position     : CHAR;

PROCEDURE Tab (VAR Phrase : Mots; Champ : INTEGER; Pos : CHAR);
              {c'est la seule ligne modifiée}
VAR
   EspacesG, EspacesD : STRING[80];
   I, EspacesCoteG, EspacesCoteD, EspacesSurLigne : INTEGER;
BEGIN
   EspacesG := '';
   EspacesD := '';
   EspacesSurLigne := Champ - LENGTH (Phrase);

   IF (Pos = 'C') OR (Pos = 'c') THEN  {centré}
   BEGIN
      EspacesCoteG := EspacesSurLigne DIV 2;
      EspacesCoteD := (EspacesSurLigne DIV 2) + (EspacesSurLigne MOD 2);
      FOR I :=  1 TO EspacesCoteG DO EspacesG := EspacesG + '_';
      FOR I :=  1 TO EspacesCoteD DO EspacesD := EspacesD + '_';
   END;
   IF (Pos = 'D') OR (Pos = 'd') THEN  {cadré à droite}
   BEGIN
      EspacesCoteG := EspacesSurLigne;
      FOR I :=  1 TO EspacesCoteG DO EspacesG := EspacesG + '_';
   END;
   IF (Pos = 'G') OR (Pos = 'g') THEN  {cadré à gauche}
   BEGIN
      EspacesCoteD := EspacesSurLigne;
      FOR I :=  1 TO EspacesCoteD DO EspacesD := EspacesD + '_';
   END;
   WRITE (LST, EspacesG, Phrase, EspacesD);
END;

BEGIN
   BEGIN
       WRITELN ('Tapez une phrase :');
       READLN (EntreePhrase);
       WRITELN ('Largeur d une colonne :');
       READLN (Taille);
       WRITELN ('Gauche, Droite ou Centrée ?');
       READLN (Position);
       Tab (EntreePhrase, Taille, Position); {passage des paramètres}
       WRITELN (LST);
   END;
END.
```

Figure 3.43 : Programme de la Figure 3.40 modifié pour un passage de paramètres plus souple.

de tabulation vers une imprimante, un fichier disque ou un écran. Dans ce cas, on remarque que la variable Phrase est passée à la procédure Tab sans espaces et qu'elle est renvoyée par la procédure avec des chaînes d'espaces qui lui sont concaténées. On remarque également que le va-et-vient d'informations se fait *sans* avoir à déclarer les mêmes identificateurs pour le corps principal et pour la procédure.

Pour clarifier l'emploi de variables destiné au déplacement de données vers et à partir d'une fonction, on remarque dans la Figure 3.40 que la procédure et le corps principal utilisent les identificateurs Phrase et Champ alors que dans la Figure 3.41 ils font partie de la procédure Tab. La seconde méthode pourrait être employée dans n'importe quel programme, car elle n'implique pas la connaissance des identificateurs utilisés par les programmes d'appel. Le programme de la Figure 3.43 montre une procédure permettant de communiquer dans les deux sens avec le programme d'appel. Le programme principal emploie l'identificateur EntréePhrase comme phrase entrée (sans espaces) et l'utilise également pour contenir la phrase modifiée (texte + espaces). Par contre, la procédure TAB ne se sert jamais de cet identificateur. En fait, la dernière ligne de la procédure Tab semble employer son propre *identificateur défini localement* (Phrase) pour recevoir cette information.

En quelque sorte, nous avons établi un lien entre Phrase, connue uniquement de la procédure Tab, et EntréePhrase, connue uniquement du programme d'appel. Cette prouesse a été réalisée en ajoutant simplement le mot réservé VAR à la première ligne de la procédure Tab.

Lors de l'utilisation du mot réservé VAR, le Turbo a exécuté une assignation entre Phrase et EntréePhrase. Il n'a pas été nécessaire de reproduire les définitions dans les deux endroits ou d'écrire explicitement la ligne :

```
Phrase = EntreePhrase
```

N'importe quel programme d'appel utilisant une variable quelconque déclarée par une chaîne de 80 caractères peut se servir de cette procédure. Le compilateur Turbo prend en charge l'équivalence entre la chaîne entrée et la chaîne que la procédure Tab emploie pour des manipulations internes. De même, il se charge de l'équivalence entre Phrase, après sa modification par la procédure Tab, et l'identificateur de variable utilisé initialement par

le programme d'appel pour passer l'information à la procédure.

On exécute le programme de la Figure 3.43 et on remarque qu'il envoie ses résultats sur l'écran pour faciliter la mise au point. Pour l'imprimer, il suffit de remplacer ou de modifier la dernière instruction WRITELN :

```
WRITELN (LST, EntreePhrase) ;
```

On aurait également pu l'écrire dans un fichier ou envoyer les résultats à la fois sur l'écran et sur l'imprimante. La procédure Tab est maintenant plus souple ; elle prend d'abord le paramètre Phrase, puis le paramètre Champ et enfin le paramètre Pos sans s'occuper des identificateurs que le programme d'appel peut utiliser pour les mêmes informations. En fait, on pourrait employer dans diverses parties du programme d'appel des identificateurs de phrase différents ou même aucun identificateur ; la procédure Tab fonctionnerait aussi bien. L'entrée suivante peut être traitée correctement par cette procédure :

```
Tab (NombreDeTickets, 20, D) ;
```

On peut également afficher les informations de l'un des exemples de programme précédents :

```
Tab (NomMembresPersonnel, TailleColonne, G) ;
```

NomMembresPersonnel et TailleColonne sont évidemment des identificateurs propres au programme d'appel. L'instruction suivante n'est pas valide :

```
Tab ('SECTION 3 : TECHNIQUE COURANTE', 80, C)
```

Le programme d'appel utilise directement une chaîne et non une variable de chaîne, c'est-à-dire que la phrase n'est pas connectée à un identificateur. Pour le Turbo qui considère un identificateur comme étant une adresse mémoire, la procédure Tab ne sait pas où rechercher la chaîne.

En résumé, nous avons indiqué à la procédure Tab que la première valeur est une chaîne de caractères (inférieure ou égale à 80) qu'elle doit appeler Phrase, la deuxième valeur est un entier qu'elle doit appeler Champ et la dernière est un caractère unique qu'elle doit appeler Pos. En ajoutant le mot réservé VAR à cette

liste de spécifications de données, nous avons indiqué à la procédure qu'elle était autorisée à modifier cette information et pas uniquement à la lire. Lorsque la procédure Tab a ajouté tous les espaces aux caractères initiaux, le Turbo assigne la valeur de Phrase (à partir de la procédure Tab) à EntréePhrase (dans le programme d'appel). Cela est le processus inverse mais également automatique du processus d'assignation de la valeur d'Entrée Phrase à Phrase au début de la procédure Tab. Cette procédure prend simplement ses valeurs d'entrée et modifie l'une d'entre elles (Phrase).

Dans le programme initial (Figure 3.40), nous avons employé un seul jeu d'adresses mémoire pour stocker la chaîne Phrase. Dans le programme de la Figure 3.41, nous en avons utilisé deux pour stocker des copies de la même chaîne ; une adresse mémoire sert à stocker EntréePhrase. La procédure Tab peut lire cette information mais n'a pas le droit de la modifier. Elle doit employer deux adresses mémoire supplémentaires dans lesquelles elle copie le contenu d'EntréePhrase puis opère sur cette copie. Ce processus protège l'entrée initiale d'une éventuelle modification mais nécessite une taille mémoire plus importante, puisque chaque variable passée à la fonction existe en mémoire à deux endroits différents.

Cependant, quand on utilise les paramètres de variables dans la Figure 3.43, on emploie une seule adresse mémoire qui contient une variable (la phrase entrée) et sert également au stockage du résultat de la procédure. Lorsque l'on passe des nombres simples ou des phrases courtes, cela est un point d'intérêt mineur ; mais lorsqu'il s'agit de tableaux de données importants, l'utilisation d'un paramètre de variable économise l'espace mémoire qui aurait été consacré à la création d'une "copie de travail " supplémentaire du tableau. Il faut également ajouter que cette économie a aussi une incidence sur la rapidité d'exécution du programme.

La capacité d'exploiter les procédures et les fonctions est la clé de la maîtrise du Turbo. Au fur et à mesure qu'il acquiert de l'expérience en matière de programmation, l'utilisateur peut développer

---

Important : Le passage de chaîne à une procédure comporte une particularité. Si la chaîne passée à partir du programme d'appel n'a pas été définie comme ayant *exactement la même longueur que la chaîne de la procédure*, le Turbo génère un message d'erreur. Heureusement, il propose une manière de résoudre ce problème : on inclut une directive de compilation ({$V–}) pour indiquer que cette situation doit être ignorée.

---

des groupes de fonctions réalisant des opérations dont il a couramment besoin : traitement de texte, formatage de pages, résolution de problèmes de navigation, etc. Le développement de ce type de programmes personnalisés ne demande que peu de travail supplémentaire et il n'est pas nécessaire de le réitérer chaque fois. C'est un peu comme s'il écrivait son propre langage de programmation sans avoir à apprendre les notions théoriques relatives à la conception du compilateur.

Fonctions créées par l'utilisateur

Tout ce qui a été dit au sujet des procédures s'applique également aux fonctions. Les fonctions peuvent contenir des variables locales mais, à l'inverse des procédures, qui renvoient ou non des informations au programme d'appel, les fonctions, par définition, doivent lui renvoyer une ou plusieurs valeurs. Ce point est illustré par la liste des fonctions prédéfinies de la Figure 3.39. Par exemple, on appelle la fonction ABS, on lui attribue un argument, puis elle renvoie la valeur absolue de cet argument. De même, on appelle la fonction SQR et on lui attribue un argument, puis elle renvoie sa racine carrée.

Les fonctions font partie des caractéristiques les plus puissantes du Turbo. Comme nous l'avons mentionné précédemment, il est même possible de redéfinir et d'adapter ces fonctions prédéfinies pour des applications particulières. La plupart du temps, l'utilisateur a besoin de créer une fonction personnelle pour résoudre un problème courant. Dans la Figure 3.44, le programme de démonstration appelle une fonction qui calcule un intérêt composé. C'est un exemple de type de codage qu'il n'est pas nécessaire de réécrire pour chaque programme financier.

On exécute plusieurs fois le programme avec différentes valeurs. La Figure 3.45 montre le résultat d'un exemple de déroulement du programme.

Le programme d'appel ne perd pas d'espace mémoire et ne nécessite pas de déclaration pour des informations passées directement à une fonction, sauf (comme avec A, B ou C) si on a une raison légitime de stocker cette information (dans cet exemple, pour une interaction avec l'écran) dans le programme. Une fois ce point assimilé, on peut modifier le programme pour qu'il n'affiche pas la démonstration. On remarque que le résultat de la fonction

```
PROGRAM Fonction_calculant_des_interets_composes;

VAR
   A, B, C, Montant : REAL;

FUNCTION Interet (Principal, Terme, Taux : REAL) : REAL;
VAR
   UnAnMultipli, PlusAnMultipli,InteretPourcent : REAL;

BEGIN
   InteretPourcent := Taux / 100.0;
   UnAnMultipli := InteretPourcent + 1.0;
   PlusAnMultipli := EXP(Terme *InteretPourcent);
   Interet := Principal * PlusAnMultipli;
END;                                   {fin de la fonction intérêt}

BEGIN                                  {programme principal}
   WRITELN;
   WRITELN (Interet (1000, 3, 7.5) :5:2);
   WRITELN ('Le 1er exemple envisage 1000 frs placés à 7,5% durant 3 ans');
   WRITELN; WRITELN;
   WRITELN ('Le second exemple attend des entrées et génère un résultat.');

          {On remarque que les variables A, B et C ne sont pas utilisées
           par la fonction !!!}

   WRITELN;
   WRITELN ('Montant principal :');
   READLN (A);
   WRITELN ('Terme en années :');
   READLN (B);
   WRITELN ('Taux d intérêt :');
   READLN (C);
   WRITELN ('Le résultat est :');
   WRITELN (Interet (A, B, C) :6:2);
END.
```

Figure 3.44 : Programme de démonstration d'une fonction calculant l'intérêt.

appelée Intérêt peut être traité comme une valeur stockée dans l'identificateur appelé Intérêt.

La fonction Intérêt se sert d'une fonction standard prédéfinie, la fonction exponentielle EXP.

On remarque également que, dans la dernière instruction WRITELN de la procédure, on a un bon exemple de fonction employée à la place d'une variable.

Si Argent a été déclaré comme une variable réelle et si on utilise les valeurs de la Figure 3.45, les deux parties de programme sui-

vantes sont équivalentes et génèrent un résultat identique :

WRITELN (Interet (A,B,C) :6 :2) ;
Argent : = 81337.92 ;
WRITELN (Argent :6 :2) ;

```
1252.32
Le 1er exemple envisage 1000 frs placés à 7,5% durant 3 ans


Le second exemple attend des entrées et génère un résultat.

Montant principal :
3000
Terme en années :
20
Taux d intérêt :
16.5
Le résultat est :
81337.92
```

Figure 3.45 : Résultat du déroulement du programme de la Figure 3.44.

Eléments supplémentaires relatifs aux paramètres

Les paramètres peuvent prendre toutes les formes possibles, y compris celles des structures complexes et de types de données créés par l'utilisateur. Cependant, à l'inverse du Pascal Standard, le Turbo n'autorise pas le passage d'une procédure ou d'une fonction à une autre procédure ou fonction comme paramètre. Bien que certains programmeurs Pascal considèrent que c'est une faiblesse du Turbo, ce langage offre une telle souplesse qu'il est toujours possible de contourner cette difficulté.

Il faut déclarer les procédures et les fonctions avant de les appeler. Dans les exemples de programmes, on a défini les codes de Tab et d'Intérêt avant de les employer. Parfois, on ne veut pas encombrer le début d'un programme avec des procédures à utiliser

ultérieurement. Plus couramment, on a besoin d'une procédure supplémentaire qui n'a pas été initialement planifiée. Dans ces deux cas, le Turbo permet de placer la procédure à l'endroit désiré. Cependant il faut insérer, quelque part avant le module dans lequel la procédure est appelée, une *référence éloignée* (FORWARD) qui indique au Turbo de rechercher le code de procédure dans l'ensemble du programme.

On modifie le programme de la Figure 3.44 pour qu'il soit identique à celui de la Figure 3.46. Nous avons alors un programme

```
PROGRAM Fonction_calculant_des_interets_composes;
VAR
   A, B, C, Montant : REAL;
   Iterations : INTEGER;

PROCEDURE Demo;
BEGIN
   WRITELN;
   WRITELN ('Montant principal :');
   READLN (A);
   WRITELN ('Terme en années :');
   READLN (B);
   WRITELN ('Taux d intérêt :');
   READLN (C);
   WRITELN ('Le résultat est :');
   WRITELN (Interet (A, B, C) :6:2);
END;

FUNCTION Interet (Principal, Terme, Taux : REAL) : REAL;
VAR
   UnAnMultipli, PlusAnMultipli,InteretPourcent : REAL;

BEGIN
   InteretPourcent := Taux / 100.0;
   UnAnMultipli := InteretPourcent + 1.0;
   PlusAnMultipli := EXP(Terme *InteretPourcent);
   Interet := Principal * PlusAnMultipli;
END;                                     {fin de la fonction intérêt}

BEGIN
   FOR Iterations := 1 TO 3 DO demo;
END.
```

Figure 3.46 : Programme de démonstration d'une séquence d'appel erronée.

rincipal appelant la procédure de démonstration qui appelle à son)ur la fonction Intérêt. Bien qu'un peu artificiel pour un programme imple, cet exemple reflète l'ordre dans lequel les choses doivent tre faites dans des programmes sophistiqués. La compilation de e programme génère un message d'erreur. Pour éviter ce pro-lème courant, on utilise la référence éloignée.

Lorsque l'on essaie de compiler le programme de la Figure 3.46, ı procédure Démo ne "trouve " pas la fonction Intérêt ; celle-ci 'a pas encore été déclarée. On modifie le programme comme ur la Figure 3.47 ; sa compilation et son déroulement ne doivent oser aucun problème.

Pour certaines situations dans lesquelles on veut relier des pro-rammes, il est plus facile de taper un nombre de références FOR-/ARD donné que de déplacer des blocs de code.

Il reste encore un détail à aborder pour les programmeurs expé-mentés. Il est possible de créer des sous-programmes en langage ssembleur et de les appeler dans un programme Turbo lors de ı compilation ; ces programmes portent le nom de *procédures xternes*. Ce processus très performant comporte de nombreuses ›strictions. Enfin, le Turbo offre la rapidité et le jeu de possibilités ormalement disponibles en langage assembleur. Les procédures t les fonctions du Turbo standard comprennent les fonctions sui-antes : générer et répondre aux interruptions du système, modifier es adresses mémoire spécifiques, effectuer des appels au BDOS ›ystème d'exploitation du disque) et au BIOS (système de gestion es entrées/sorties), communiquer avec le système de gestion des rreurs et envoyer des données aux ports d'entrées/sorties du ystème. La plupart de ces caractéristiques sont uniquement mployées par les programmeurs chevronnés. Même les pro-rammeurs expérimentés auront du mal à trouver une application ui ne puisse pas être résolue avec le Turbo et qui nécessite .bsolument le recours au langage assembleur.

```
PROGRAM Fonction_calculant_des_interets_composes;

VAR
   A, B, C, Montant : REAL;
   Iterations : INTEGER;

{référence en-avant (forward)}
FUNCTION Interet (Principal, Terme, Taux : REAL) : REAL; FORWARD;


PROCEDURE Demo;
BEGIN
   WRITELN;
   WRITELN ('Montant principal :');
   READLN (A);
   WRITELN ('Terme en années :');
   READLN (B);
   WRITELN ('Taux d intérêt :');
   READLN (C);
   WRITELN ('Le résultat est :');
   WRITELN (Interet (A, B, C) :6:2);
END;

FUNCTION Interet;
VAR
   UnAnMultipli, PlusAnMultipli,InteretPourcent : REAL;

BEGIN
   InteretPourcent := Taux / 100.0;
   UnAnMultipli := InteretPourcent + 1.0;
   PlusAnMultipli := EXP(Terme *InteretPourcent);
   Interet := Principal * PlusAnMultipli;
END;                                {fin de la fonction intérêt}

BEGIN
   FOR Iterations := 1 TO 3 DO demo;
END.
```

Figure 3.47 : Programme de démonstration d'une séquence de déclaration et d'appel de fonction correcte.

4. STRUCTURES DE DONNÉES AVANCÉES

PRÉSENTATION

Niklaus Wirth, concepteur du Pascal, pense que la programmation est l'art de combiner les algorithmes et les structures de données. Une grande partie de cet ouvrage a traité du développement des algorithmes : procédures, fonctions et contrôle du déroulement du programme avec tests et structures de contrôle. Jusqu'à présent, nous avons employé des types de données prédéfinis : caractères, entiers, nombres réels et occasionnellement une valeur booléenne. Ces types de données manipulent des entrées, des sorties et le contrôle de boucles et de tests. Tout ce qui a été exécuté sur des données aurait aussi bien pu être réalisé dans des langages tels que le BASIC. Il est temps maintenant d'explorer les outils du Turbo qui donnent un contrôle réel sur les données.

La représentation et l'organisation des données est discriminatoire et ne consiste pas uniquement à introduire des informations dans un programme. La représentation dans sa forme la plus naturelle constitue la moitié de la tâche de programmation. De plus, si cette tâche est définie comme devant résoudre des problèmes réels, les techniques de programmation les plus simples et les plus intuitives sont celles qui sont susceptibles de générer les programmes les plus efficaces. Le Turbo offre un jeu de types de données et de structures puissant pour accélérer la transformation

en données pouvant être traitées par l'ordinateur.

Les structures de données Turbo (tableaux, enregistrements et fichiers) se complètent. Lorsque l'on s'attelle à des tâches de programmation plus ambitieuses, on utilise plusieurs de ces structures dans le même programme. On remarque que presque tous les exemples de ce chapitre exécutent des tâches réelles à l'aide de caractéristiques qui n'ont pas encore été abordées. Il est possible d'ignorer ces traits nouveaux en se concentrant sur les aspects déjà étudiés, puis de revenir à ces programmes lors de l'apprentissage des autres techniques pour voir comment les différentes structures et les différentes techniques peuvent être associées.

TABLEAUX

En Turbo comme en BASIC, un tableau est une collection d'*éléments de données similaires*. On peut avoir des tableaux d'entiers, de chaînes, de nombres réels, représentant respectivement, par exemple, des scores, des noms et des prix. Un tableau est une structure naturelle qui reflète des types de listes faites dans la vie de tous les jours : notes à payer (toutes en francs), personnes à qui écrire (toutes des gens), etc. Un tableau doit être déclaré par un identificateur, type de données auquel ces éléments appartiennent, et par le numéro du premier et du dernier éléments du tableau. Voici des exemples de déclarations :

```
ListeNom            :ARRAY[1..30] OF Nom ;
ScoreTest           :ARRAY[1..10] OF REAL ;
PiecesVisitees      :ARRAY[Cave..Cuisine] OF Pieces ;
Conditions          :ARRAY[1..8] OF BOOLEAN ;
EntreeCaracteres    :ARRAY[a..z] OF CHAR ;
JoursTravail        :ARRAY[Lundi..Vendredi] OF Jours ;
```

Dans les premier, troisième et sixième exemples, les types de données Nom, Pièces et Jours doivent tous avoir été déclarés précédemment, car ils ne sont pas prédéfinis. A l'inverse du BASIC qui demande uniquement l'entrée de la taille maximale d'un tableau, le Turbo demande une valeur de départ et une valeur de fin. Cela évite la frustration de nombreux programmeurs lorsqu'ils travaillent

avec des tableaux BASIC qui commencent par 0 au lieu de 1. En fait, les tableaux peuvent être déclarés dans les termes les plus naturels selon la situation. Si l'on travaille sur un type de données appelé Jours, pourquoi ne pas dimensionner le tableau avec des éléments appelés Lundi-Vendredi plutôt que 1-5. Si l'on a créé un tableau appelé Pièces pour un jeu d'aventures qui commence dans la Cave et se termine dans la Cuisine, il est naturel de le définir de la même manière.

On accède à un élément individuel par son indice exactement comme en BASIC. Il ne faut pas oublier que les tableaux peuvent être passés à des procédures comme paramètres, technique couramment utilisée dans les procédures de tri et de conversion de code générique. La manipulation de tableaux est assez directe, car la caractéristique de contrôle des entrées du Turbo empêche le stockage de données sans signification dans le tableau. Si l'on essaie de stocker une valeur qui n'est pas du type déclaré ou d'accéder à un élément qui se trouve en dehors des valeurs d'indices maximal et minimal déclarées, le Turbo signale l'erreur immédiatement. De même, si l'on calcule une valeur d'indice à l'aide de nombres réels et que l'on essaie d'accéder à un élément de tableau avec, par exemple, l'indice calculé de 1.999999999, le Turbo signale l'erreur ; il vérifie également les indices pour s'assurer qu'il s'agit de valeurs réellement scalaires (avec un successeur et un prédécesseur immédiats).

Le Pascal standard n'accepte pas les chaînes de caractères mais autorise les tableaux. Par conséquent, dans de nombreux ouvrages relatifs au Pascal, on relève une insistance particulière sur les tableaux car ils offrent le seul moyen pratique de manipuler du texte (par exemple pour utiliser des codes de contrôle de l'écran). Le Turbo ne permet pas seulement l'emploi de chaînes de caractères ; il propose une série de procédures et de fonctions pour les traiter.

Les tableaux sont des structures de données entièrement stockées en mémoire ; par conséquent, les opérations sur les tableaux ne sont pas ralenties par le temps d'accès au disque. Ce sont les structures de données les plus rapides et les plus simples pour le tri, la gestion de répertoires et les codes de conversion. Le programme de la Figure 4.1 utilise un tableau simple pour réaliser un tri rapide. Ce programme apparaît dans des centaines d'ouvrages de programmation sous une forme ou sous une autre.

On tape le programme et on l'exécute ; il ne se contente pas d'illustrer la déclaration et l'utilisation optimale des tableaux, il

démontre également, y compris aux utilisateurs sceptiques, la rapidité du Turbo Pascal. En fait, on peut modifier le programme pour qu'il trie une très grande quantité de nombres aléatoires. Si l'on essaie d'exécuter le même programme en BASIC ou dans un autre langage et que l'on chronomètre la vitesse d'exécution, on comprend pourquoi le Turbo porte ce nom.

Bien que les algorithmes de tri dépassent l'objet de ce livre, l'utilisateur peut adapter cet exemple pour exécuter des tris rapides dans ses propres programmes d'application. De nombreux tableaux sont employés dans les exemples du reste de cet ouvrage. Le programme de la Figure 4.2 est la version Tri par bulle du programme de la Figure 4.1. Bien qu'il ne soit pas aussi rapide que le tri de Shell, il est cependant plus performant que le même processus exécuté dans un autre langage et emploie également de façon optimale des tableaux simples. En Turbo, un tri par bulle de moins de 100 valeurs combine simplicité et très grande rapidité. On l'exécute avec différentes valeurs pour MaxEléments ; comme le tri de Shell, c'est l'une des premières applications des tableaux simples.

Tableaux multidimensionnels

Les tableaux peuvent avoir plusieurs dimensions. Pour cette caractéristique, le Turbo ressemble beaucoup au BASIC. Il ne faut pas oublier cependant que tous les éléments d'un tableau doivent appartenir au même type de données. Le BASIC ayant une gamme illimitée de structures de données, de nombreux programmeurs ont pris l'habitude de déclarer la plupart de leurs tableaux comme des tableaux de chaînes et même de stocker les valeurs numériques comme des chaînes. L'exemple de tableau à deux dimensions classique contient la liste d'un professeur avec les noms des élèves et leurs résultats. En BASIC, ce type de tableau nécessite le stockage des noms *et* des résultats sous la forme de chaînes dans un tableau. Le Turbo offre une structure d'*enregistrement* beaucoup plus souple pour stocker des éléments reliés de façon logique mais de type différent. Les *tableaux d'enregistrements* servent beaucoup en Turbo ; les enregistrements sont traités dans la dernière partie de ce chapitre.

Les tableaux multidimensionnels servent souvent à stocker des tables de consultation. Il est en effet plus simple et plus rapide

```
PROGRAM Tri_de_Shell;          {tri de Shell sur 500 nombres aléatoires}
CONST
   MaxEle = 500;
VAR
   Nums : ARRAY [1..500] OF INTEGER;
   Temp, I, J, Pass, Vide : INTEGER;
PROCEDURE Entree_Gen;
BEGIN
   FOR I := 1 TO MaxEle DO
   BEGIN
      Nums[I] := RANDOM (1000);
   END;
END;
PROCEDURE Sortie;
BEGIN
   FOR I := 1 TO MaxEle DO
   BEGIN
      WRITE (Nums[I]);
      WRITE ('  ');
   END;
   WRITELN; WRITELN; WRITELN;
END;
PROCEDURE Interversion;
BEGIN
   Temp := Nums[J];
   Nums[J] := Nums[J + Vide];
   Nums[J + Vide] := Temp;
END;

PROCEDURE Tri;
BEGIN
   Vide := MaxEle DIV 2;
   WHILE Vide > 0 DO
   BEGIN
      FOR I := (Vide + 1) TO MaxEle DO
      BEGIN
         J := I - Vide;
         WHILE J > 0 DO
         BEGIN
            IF Nums[J] > Nums[J + Vide] THEN
            BEGIN
               Interversion;
               J := J - Vide;
            END
            ELSE J := 0;
         END;                   {fin de while}
      END;
      Vide := Vide DIV 2;
   END;                         {tant que Vide >0}
END;                            {Tri}
```

Figure 4.1 : Utilisation d'un tableau unidimensionnel dans un tri de Shell.

```
BEGIN
   WRITELN ('Début du programme de tri de Shell.');
   Entree_Gen;
   WRITELN ('Le tableau de ',MaxEle,' éléments est chargé et le tri commence.');
   Tri;
   WRITELN ('Le tri est terminé ; il y a ',MaxEle,' valeurs classées.');
   Sortie
END.
```

Figure 4.1 (suite)

de rechercher une valeur dans un tableau plutôt que de calculer cette valeur (surtout si la table est organisée de telle sorte que les valeurs les plus fréquemment nécessaires soient accessibles en premier). De plus, certaines tables ne se prêtent pas à l'exécution de calculs rapides. Il est presque toujours plus facile d'élaborer une table, au moins pour couvrir les valeurs les plus souvent recherchées que de calculer éternellement les mêmes prix.

Un tableau à deux dimensions (ou plus) se présente normalement sous la forme d'une grille. Par exemple, la Figure 4.5 est la représentation d'un tableau à deux dimensions généré par le programme de la Figure 4.3. Chaque élément d'un tableau à deux dimensions est identifié de façon unique par deux indices ; l'un se réfère à la rangée et l'autre à la colonne. La syntaxe de la déclaration et du dimensionnement d'un tableau contenant 21 éléments stockés sur 3 rangées et 7 colonnes est :

```
VAR
      Horaire : ARRAY [1..3, 1..7] OF INTEGER ;
```

Pour identifier un élément spécifique de ce tableau à 2 dimensions, il faut 2 indices :

```
Semaine := Taux [12, 25] ;
WeekEndArrivee := TableauTrain [3, 4]
```

L'une des applications d'un tableau à deux dimensions est la représentation sur ordinateur d'un horaire de train. Les colonnes représentent différents trains : direct, omnibus et trains spéciaux, et il est impossible de calculer les heures de départ et d'arrivée des trains car elles dépendent de variables trop nombreuses, mais il est très facile de les rechercher dans un tableau multidimensionnel. Le programme de la Figure 4.3 illustre les rudiments de ce processus.

```
PROGRAM Tri_par_Bulle;        {tri par bulle sur 500 nombres aléatoires}

CONST
   MaxEle = 500;

VAR
   Nums : ARRAY [1..500] OF INTEGER;
   Ele : INTEGER;
   Temp, I, J, Pass : INTEGER;
   Inter : BOOLEAN;

PROCEDURE Entree_Gen;
BEGIN
   FOR I := 1 TO MaxEle DO
   BEGIN
      Nums[I] := RANDOM (1000);
   END;
END;

PROCEDURE Sortie;
BEGIN
   FOR I := 1 TO MaxEle DO
   BEGIN
      WRITE (Nums[I]);
      WRITE ('  ');
   END;
   WRITELN; WRITELN; WRITELN;
END;

PROCEDURE Interversion;
BEGIN
   Temp := Nums[J];
   Nums[J] := Nums[J + 1];
   Nums[J + 1] := Temp;
   Inter := TRUE;
END;

PROCEDURE Tri;
BEGIN
   FOR I := 2 TO MaxEle DO
   BEGIN
      J := I - 1;
      WHILE J > 0 DO
      BEGIN
         IF Nums[J] > Nums[J + 1] THEN
         BEGIN
            Interversion;
```

Figure 4.2 : Utilisation d'un tableau unidimensionnel dans un tri par bulle.

```
            J := J - 1;
          END
          ELSE J := 0;
        END;                    {fin de while}
      END;
    END;                        {Tri}

    BEGIN
      WRITELN ('Début du programme de tri par bulle.');
      Entree_Gen;
      WRITELN ('Le tableau de ',MaxEle,' éléments est chargé et le tri commence.');
      Tri;
      WRITELN ('Le tri est terminé ; il y a ',MaxEle,' valeurs classées.');
      Sortie
    END.
```

Figure 4.2 (suite)

Le Turbo permet la création de tableaux à plus de deux dimensions ; malheureusement, à l'inverse de la plupart de ses structures de données, ceux-ci ne sont pas un simple reflet de l'organisation des informations dans le monde réel. Ils sont souvent employés dans des jeux d'aventures pour représenter des structures complexes (mais pas très réalistes) telles que des labyrinthes qui peuvent contenir simultanément des trésors, des pièges, des monstres, et le joueur. Il existe plusieurs procédures de manipulation de chaînes utilisées dans le programme de la Figure 4.3 pour charger le tableau. On peut les ignorer pour l'instant et y revenir après la lecture du paragraphe concernant les chaînes. La Figure 4.4 montre le résultat généré par le déroulement du programme.

Le but de ce programme est d'illustrer une application de tableau au stockage des données qui se prêtent à un format tabulaire. Dans une version définitive, on chargerait probablement les informations dans le tableau à partir d'un fichier de données externe ou on utiliserait une procédure plus sophistiquée (peut-être avec des boucles imbriquées). On voit dans ce programme que les tableaux multidimensionnels ne sont pas adaptés à l'interrogation interactive à cause de l'ensemble des traitements que les réponses de l'utilisateur doivent subir pour être converties en données utilisables pour l'attribution d'indices relatifs à une donnée du tableau.

On remarque combien les messages doivent être complexes

```
PROGRAM Horaire_Train;

CONST
   Limite = 7;               {nombre maximum de trains dans cet horaire}

TYPE
   Trains = STRING[10];

VAR
   Table_Train : ARRAY[1..3, 1..7] OF Trains; {tableau de 3 colonnes, 4 rangées}
   Rang, Col : INTEGER;

PROCEDURE Charge_Tableau;
CONST
{Toutes les informations relatives aux horaires de train sont contenues dans
ces 3 lignes. Les chaines lisent 10 caractères à la fois qui sont ensuite
assignés au tableau. La procédure utilisant un nombre déterminé de caractères,
des espaces sont ajoutés aux données pour constituer des groupes de 10
caractères}

   RangTrain = 'TGV MéditeTGV AtlantMistral   Capitole Corail SudCorail EstTrans-sibé';
   RangDep   = '08:00     09:15     10:03     14:37     20:00     21:18     23:55     ';
   RangArr   = '08:13     09:30     10:24     14:52     20:12     21:31     00:07     ';
VAR
   Prem_Car :INTEGER;
BEGIN
   Prem_Car := 1;
   Col := 1;
   FOR Rang := 1 TO Limite DO
   BEGIN
      Table_Train [Col, Rang] := COPY (RangTrain, Prem_Car, 10);
      Prem_Car := Prem_Car + 10;
   END;
   Prem_Car := 1;
   Col := 2;
   FOR Rang := 1 TO Limite DO
   BEGIN
      Table_Train [Col, Rang] := COPY (RangDep, Prem_Car, 10);
      Prem_Car := Prem_Car + 10;
   END;
   Prem_Car := 1;
   Col := 3;
   FOR Rang := 1 TO Limite DO
   BEGIN
      Table_Train [Col, Rang] := COPY (RangArr, Prem_Car, 10);
      Prem_Car := Prem_Car + 10;
   END;
```

Figure 4.3 : Programme de démonstration de l'utilisation d'un tableau multidimensionnel.

```
TRAIN          DEPART    ARRIVEE
--------------------------------

TGV Médite     08:00     08:13
TGV Atlant     09:15     09:30
Mistral        10:03     10:24
Capitole       14:37     14:52
Corail Sud     20:00     20:12
Corail Est     21:18     21:31
Trans-sibé     23:55     00:07
```

Figure 4.4 : Résultat généré par le résultat du programme de la Figure 4.3.

et explicites pour guider les deux valeurs entrées par l'utilisateur. Cela ne veut pas dire qu'il est impossible d'employer des tableaux multidimensionnels pour une entrée interactive, mais leur emploi implique souvent une programmation "lourde" comprenant une série de messages spécifiques élaborés. Le programme doit vérifier qu'il a reçu des réponses entières autorisées à tous les messages, puis les combiner pour indexer une valeur.

Néanmoins, le tableau multidimensionnel est une structure parfaite pour accéder à des informations à l'intérieur d'un programme.

La Figure 4.5 est un diagramme du tableau de données employé dans les programmes des Figures 4.3 et 4.4. Il faut essayer différentes entrées jusqu'à la compréhension du mode de stockage des données.

CHAÎNES ET CARACTÈRES

Dans une réunion d'informaticiens, la conversation porte tôt ou tard sur l'écriture de compilateurs et de programmes d'analyse syntaxique. Bien que le compilateur reste le fief des meilleurs programmeurs, il semble qu'un jour ou l'autre tous les programmeurs ont à écrire des programmes d'analyse syntaxique. Ces programmes interprètent des entrées ; par exemple, dans un jeu d'aventures, les réponses telles que Nord ou simplement N sont interprétées par un programme d'analyse syntaxique qui convertit ces entrées tapées sur le clavier en données à partir desquelles le programme peut déterminer une nouvelle position.

	Colonne 1	Colonne 2	Colonne 3
Rangée (1)	TGV Médite	08:00	08:13
(indices)	1,1	2,1	3,1
Emplacement	Col 1, Rang 1	Col 2, Rang 1	Col 3, Rang 1
Rangée (2)	TGV Atlant	09:15	09:30
	1,2	2,2	3,2
	Col 1, Rang 2	Col 2, Rang 2	Col 3, Rang 2
Rangée (3)	Mistral	10:03	10:24
	1,3	2,3	3,3
	Col 1, Rang 3	Col 2, Rang 3	Col 3, Rang 3
Rangée (4)	Capitole	14:37	14:52
	1,4	2,4	3,4
	Col 1, Rang 4	Col 2, Rang 4	Col 3, Rang 4
Rangée (5)	Corail Sud	20:00	20:12
	1,5	2,5	3,5
	Col 1, Rang 5	Col 2, Rang 5	Col 3, Rang 5
Rangée (6)	Corail Est	21:18	21:31
	1,6	2,6	3,6
	Col 1, Rang 6	Col 2, Rang 6	Col 3, Rang 6
Rangée (7)	Trans-sibé	23:55	00:07
	1,7	2,7	3,7
	Col 1, Rang 7	Col 2, Rang 7	Col 3, Rang 7

Figure 4.5 : Représentation d'un tableau multidimensionnel généré par le programme de la Figure 4.3.

La tâche de ce type de programme peut être aussi simple que la conversion d'un choix de menu composé d'un seul chiffre ou aussi complexe que l'interprétation d'une commande telle que “METTRE LE DIAMANT ET L'OEUF DANS LA CASE TRESOR ”. Les programmes d'analyse syntaxique ont une caractéristique commune : ils doivent analyser et tester des chaînes. L'écriture de tels programmes est amusante, car ils acceptent une gamme d'entrées très large et non une liste rigide de codes arbitraires.

Les commandes de chaînes se prêtent également à l'écriture de programmes de communication, de traitement de texte et d'une série d'utilitaires pour la conversion de données. Comme nous l'avons mentionné précédemment, les chaînes n'existent pas en Pascal standard et leur absence a longtemps été considérée comme sa plus grande faiblesse. Le Turbo remédie à cet inconvénient.

Nous avons déjà vu des exemples de chaînes dans le programme d'horaire de train et dans le programme de tabulation du Chapitre 3. Il est temps maintenant de donner une définition formelle. Une *chaîne* est une liste séquentielle de caractères ; en Turbo, c'est un type de données prédéfinies avec une longueur maximale de 255 caractères. On remarque que les chaînes constituent un type de données *structuré* (ou *complexe*) car elles sont composées d'éléments d'un type de données plus fondamental, le caractère. Les chaînes doivent être déclarées comme n'importe quelles autres données. Chacune d'entre elles a un identificateur et une taille maximale. Voici des exemples de déclarations :

```
TYPE
      Nom = STRING [20] ;
VAR
      Tampon : STRING [127] ;
```

On remarque que les chaînes sont déclarées à la fois comme un type de données (à l'aide d'un signe égal) et comme une variable (signe deux-points, :). En fait, étant donné que l'on peut uniquement assigner des valeurs à des variables et non à des types, si l'on déclare une chaîne comme un type de données, il faut également déclarer au moins une variable de ce type pour contenir toutes les valeurs. Voici un exemple de déclaration :

```
TYPE
      MessageTampon = STRING [132] ;
VAR
      EntreeModem, SortieModem : MessageTampon ;
```

Dans ce cas, EntréeModem et SortieModem sont des variables qui peuvent contenir des données. Ce sont toutes deux des chaînes dont la taille ne doit pas excéder 123 caractères.

Le plus souvent, les chaînes sont employées comme des variables. Tout ce qui est exécuté avec un tableau de caractères unidimensionnel peut l'être avec une chaîne. On remarque également que si l'on doit déclarer une taille maximale pour une variable de chaîne, celle-ci peut contenir des chaînes plus courtes. Le Turbo les complète automatiquement par des espaces. Des commandes spécifiques permettent de déterminer la longueur d'une chaîne (seulement les caractères, pas les espaces), d'extraire des sous-chaînes de n'importe quelle longueur à partir d'une position quelconque, de concaténer des chaînes, de les convertir en leur valeur numérique, de les comparer, etc. Étant donné que les chaînes sont des tableaux de caractères, le Turbo permet de manipuler et d'accéder à des caractères individuels comme s'il s'agissait d'éléments d'un tableau.

Les procédures et les fonctions de manipulation de chaînes apparaissent Figure 4.6.

La Figure 4.7 montre un petit programme illustrant les commandes de manipulation de chaînes ; les opérations sont commentées dans les instructions. On peut essayer différents paramètres dans les commandes ou dans les chaînes, notamment les paramètres générateurs d'erreurs concernant le dépassement de la longueur autorisée, et essayer de rechercher des chaînes non existantes.

La Figure 4.8 reprend une version simplifiée de la procédure de chargement de tableau du programme d'horaire de train ; une chaîne est divisée en plusieurs parties à stocker dans un tableau unidimensionnel. Les données entrées ont été modifiées avec des chiffres identiques (11 : 11 pour le premier 22 : 22 pour le deuxième, etc.) pour clarifier l'entrée et la sortie. L'écran généré par le déroulement du programme apparaît Figure 4.9.

Les abréviations de cette liste sont utilisées pour représenter les arguments des fonctions suivantes :

C	Caractère
E	Valeur entière ou valeur réelle
I	Entier
POS	Position dans une chaîne
ST1, ST2	Chaîne de caractères
SUB	Sous-chaîne de caractères

Procédures et fonctions

CONCAT (ST..ST)	Concatène un nombre quelconque d'expressions de chaîne séparées par des virgules ; le résultat est toujours une chaîne. La même fonction peut être réalisée en utilisant le signe plus comme opérateur de concaténation (par exemple, ST4 := ST1 + ST2 + ST3).
COPY (ST, POS, I)	Retourne une sous-chaîne, de la chaîne ST, de I caractères de long et commençant à partir de la position POS ; le résultat est toujours une chaîne.
DELETE (ST, POS, I)	Supprime une sous-chaîne, dans la chaîne ST, de I caractères de long et commençant à partir de la position POS ; le résultat est toujours une chaîne.
INSERT (SU, ST, POS)	Insère la chaîne SU dans la chaîne ST à partir de la position POS.
LENGTH (ST)	Retourne le nombre de caractères de la chaîne ST ; le résultat est toujours un entier.
POS (ST1, ST2)	Retourne la position du début de la chaîne ST1 dans la chaîne ST2 ; le résultat est toujours un entier.

Figure 4.6 : Procédures et fonctions de manipulation de chaînes en Turbo Pascal.

STR (E, ST)	Convertit la valeur numérique E en une chaîne ST ; E doit être un paramètre d'écriture entier ou réel (par exemple, Age : 5 pour un entier ou Francs : 5:2 pour un réel).
VAL (ST, E, I)	Convertit l'expression de chaîne ST en une valeur E entière ou réelle. Si une erreur apparaît, I contient la position du caractère qui a généré l'erreur.

```
PROGRAM Demo_Chaines;
CONST
   Phrase1 = 'Gilbert & Sullivan';
   Phrase2 = 'représente H.M.S. PINAFORE';
   Phrase3 = 'Sullivan';
   Date = '1885';
VAR
   Nombre, Taille, Place, CodeErreur : INTEGER;
   NombreReel : REAL;
   SousPhrase : STRING[80];

BEGIN
   WRITELN ('Phrase1 = ',Phrase1);
   WRITELN ('Date = ',Date);
   WRITELN;
   Taille := LENGTH (Phrase1);
   WRITELN ('La phrase1 à une longueur de : ',Taille,' caractères');
   WRITELN;
   Place := POS ('&', Phrase1);
   WRITELN ('Le & occupe la position ',Place,' dans la phrase : ',Phrase1);
   Place := POS ('G',  Phrase1);
   WRITELN ('Le G occupe la position ',Place,' dans la phrase : ',Phrase1);
   WRITELN;
   SousPhrase := COPY (Phrase1, 11, 8);
   WRITELN ('La sous-phrase comprise entre la 11ième et 18ième lettre est :');
   WRITELN (SousPhrase);
   WRITELN;
   WRITELN ('Phrase1 = ', Phrase1);
   WRITELN ('Phrase2 = ', Phrase2);
   WRITELN (CONCAT(Phrase1, Phrase2));
   WRITELN (Phrase1 + Phrase2);        {plus simple que la ligne précédente}
   WRITELN;
   WRITELN ('La date est stockée sous forme de chaîne : ',Date);
   WRITELN ('On ne peut pas exécuter d opérations mathématiques sur une chaîne.');
   VAL (Date, Nombre, CodeErreur);
   VAL (Date, NombreReel, CodeErreur);
   WRITELN ('Conversion de la date en nombre entier : ', Nombre);
   WRITELN ('Conversion de la date en nombre réel   : ', NombreReel);
   WRITELN ('La date + 25 correspond à : ', Nombre + 25);
END.
```

Figure 4.7 : Programme de démonstration d'opérations de manipulation de chaînes élémentaires.

```
PROGRAM Manipulation_de_chaine;

CONST
   Limite = 5;

TYPE
   Trains = STRING[5];

VAR
   Index : INTEGER;
   TableTrain : ARRAY [1..5] OF Trains;

PROCEDURE Charge_Table;
CONST
   RangeeDonnees = '11:1122:2233:3344:4455:5566:6677:77';
VAR
   PremCar : INTEGER;
BEGIN
   PremCar := 1;
   FOR Index := 1 TO LIMITE DO
   BEGIN
      TableTrain [Index] := COPY (RangeeDonnees, PremCar, 5);
      PremCar := PremCar + 5;
   END;
END;

PROCEDURE Sortie;
BEGIN
   FOR Index := 1 TO Limite DO
   WRITELN ('TableTrain[',Index,'] = ',TableTrain[Index]);
END;

BEGIN
   Charge_Table;
   Sortie
END.
```

Figure 4.8 : Programme de démonstration des procédures de chaînes pour charger un tableau.

```
TableTrain[1] = 11:11
TableTrain[2] = 22:22
TableTrain[3] = 33:33
TableTrain[4] = 44:44
TableTrain[5] = 55:55
```

Figure 4.9 : Écran généré par le déroulement du programme de la Figure 4.8.

Le programme de la Figure 4.8 utilise la procédure COPY. Celle-ci extrait une sous-chaîne de longueur quelconque d'une autre chaîne à partir d'une position déterminée. La syntaxe de cette procédure est identique à celle utilisée par la procédure DELETE. Pour l'emploi de l'une ou de l'autre, il faut donner au Turbo le nom de la chaîne *cible* ainsi que la longueur et la position de départ de la sous-chaîne à l'intérieur de cette chaîne. La procédure COPY copie cette sous-chaîne dans une autre chaîne tandis que la procédure DELETE l'efface. Une autre procédure très proche, INSERT, insère une chaîne à la position indiquée à l'intérieur de la chaîne cible. Cette opération ne doit pas être confondue avec la concaténation qui relie la fin d'une chaîne au début de la suivante.

Dans l'exemple ci-dessous, on assigne à la chaîne de 30 caractères EntréePhrase la valeur :

FORTRAN, BASIC et Turbo Pascal

On utilise la procédure COPY pour extraire la phrase :

Turbo Pascal

et on assigne cette phrase à une variable de chaîne appelée MeilleurLangage. Pour cela, on indique à la procédure COPY l'identificateur de la chaîne cible, la position à l'intérieur de la cible du premier caractère de la chaîne à extraire et sa longueur :

```
VAR
    EntréePhrase : STRING [30] ;
    MeilleurLangage : STRING [12] ;
BEGIN
    EntréePhrase := 'FORTRAN, BASIC et TurboPascal' ;
    MeilleurLangage := COPY (EntréePhrase,19, 12) ;
END.
```

Cependant, dans la plupart des cas, le point de départ et la longueur de la chaîne doivent être calculés à l'aide d'autres procédures.

Dans le programme de la Figure 4.8, la "chaîne cible " s'appelle RangéeDonnées, le point de départ PremCar et la longueur de la chaîne à extraire est donnée par la constante 5. Pour voir comment fonctionne la manipulation de chaînes, on modifie la chaîne RangéeDonnées ainsi que PremCar et la constante.

L'exemple suivant est un programme destiné à résoudre un problème courant : la conversion en divers formats de données. Il sert également d'introduction à la notion de fichier qui est abordée un peu plus loin dans ce chapitre. Nous avons par exemple un long fichier de données composé d'adresses de destinataires qui a été créé sur un ordinateur Amstrad à l'aide du BASIC Amstrad. La Figure 4.10 montre cette liste dans le format BASIC.

```
"Michut","Melle Adrienne","67, rue Ratounet","Les Arcs","73124"
"Lemercier","M. Marcel","14, av des pré hauts","Paris","75016"
"Brécat","M. Jacques","116, impasse des fleurs","Orsay","91160"
"Choupinet","Mme Yvonne","3, rue Skoff","Kremlin-bicètre","94789"
"Lagave","M. René","34, bd Ycelle","Gratte/mer","33008"
"Sonai","M. Simon","44, rue Barbe","Antony","92360"
```

Figure 4.10 : Exemple de fichier de données créé sur IBM PC avec le BASIC Amstrad.

On remarque comment le BASIC ajoute des virgules et des guillemets (non obligatoires) pour séparer les chaînes. Nous voulons convertir l'ensemble du fichier en une centaine de noms dans un fichier Turbo composé d'enregistrements. La Figure 4.11 donne un programme de conversion. Comme la plupart des utilitaires, il est loin d'être élégant, mais il n'y a pas vraiment de raison d'affiner les tests, d'éliminer des lignes de code facultatives et de développer une convivialité si le programme ne doit être utilisé qu'une seule fois.

Les programmeurs professionnels risquent de frémir à la vue de certaines des constructions employées dans ce programme, mais il a été écrit et mis au point rapidement et il fonctionne. Plus important encore, il illustre une application réelle et non un exemple artificiel. Nous avons dû traiter un ensemble complexe de guillemets précédés ou non de virgules.

```
PROGRAM Conversion_Fichiers_basic_Fichiers_Turbo;

TYPE
   EnregPers = RECORD
      Nom : STRING[13];
      Prenom : STRING[15];
      Rue : STRING[30];
      Ville : STRING[19];
      CodePost : STRING[5];
   END;

VAR
   FichSortie : FILE OF EnregPers;
   Pers : EnregPers;
   UnCar : CHAR;
   FichEntr : FILE OF CHAR;
   Entrzie, Sortzie : STRING[14];      (stocke les noms de fichiers)
   Reste, Tampon, Information : STRING[110];
   Premguill, Index, Guillsuiv, Compteur, LongTamp : INTEGER;

PROCEDURE Initialise;
BEGIN
   WRITE ('Nom du fichier BASIC : ');
   READLN (Entrzie);
   ASSIGN (FichEntr, Entrzie);
   RESET (FichEntr);
   WRITELN ('Nom du fichier Turbo :');
   READLN (Sortzie);
   ASSIGN (FichSortie, Sortzie);
   REWRITE (FichSortie);
   Tampon := '';
END;

PROCEDURE TraitTampon;
BEGIN
   LongTamp := 2;
   Index := 1;
   WHILE LongTamp > 1 DO
   BEGIN
      PremGuill := POS ('"', Tampon);
      LongTamp := LENGTH (Tampon);
      Reste := COPY (Tampon, PremGuill + 1, LongTamp);
      GuillSuiv := POS ('"', Reste);
      Information := COPY (Reste, 1, GuillSuiv - 1);
      IF LENGTH (Information) <> 1 THEN
      BEGIN
         CASE Index OF
```

Figure 4.11 : Programme de conversion d'un fichier de données BASIC Amstrad en format turbo.

```
            1: Pers.Nom := Information;
            2: Pers.Prenom := Information;
            3: Pers.Rue := Information;
            4: Pers.Ville := Information;
            5: Pers.CodePost := Information;
          END;
          Index := Index + 1;
        END;
        Tampon := Reste;
      END;
      Tampon := '';
      WRITE (FichSortie, Pers);
END;

PROCEDURE Lecture;
BEGIN
   WHILE NOT EOF (FichEntr) DO
      BEGIN
         READ (FichEntr, UnCar);
         Tampon := Tampon + UnCar;
         IF UnCar = #13 THEN TraitTampon;
      END;
END;

BEGIN                                    {programme principal}
   Initialise;
   Lecture;
   CLOSE (FichEntr); {CLOSE est décrite dans la section relative aux ficniers}
   CLOSE (FichSortie);
   WRITELN ('Le fichier a été traité avec succès !');
END.
```

Figure 4.11 (suite)

La procédure TraitTampon ne se contente pas d'utiliser la procédure COPY pour extraire une chaîne mais emploie également la fonction LENGTH pour déterminer la longueur d'une chaîne (en caractères, sans inclure les espaces ajoutés pour le remplissage) et la fonction POS (position). Cette dernière cherche dans une chaîne la première occurrence d'un caractère spécifique et renvoie la position (l'indice dans un tableau) de ce caractère dans la chaîne cible. Comme dans cet exemple, on emploie rarement la fonction seule, mais plutôt comme un test pour trouver une valeur employée ensuite par une autre commande telle que COPY. Dans la procédure TraitTampon, nous avons utilisé POS pour trouver les guillemets qui séparent les chaînes des fichiers BASIC, puis nous avons employé la procédure COPY pour copier le groupe de caractères texte (dont la longueur correspond à la distance entre les guillemets) suivant :

Information := COPY (Reste,1,GuilleSuiv−1)

Une fois la notion de fichier Turbo assimilée, on peut revenir à ce programme et faire un fichier test identique à celui de la Figure 4.10 pour s'entraîner. On modifie la procédure TraitTampon comme sur la Figure 4.12 pour afficher les variables les unes après les autres. On remarque la ligne :

DELAY (400)

DELAY est une procédure prédéfinie du Turbo qui génère une attente correspondant au nombre de millisecondes spécifié. Elle sert beaucoup à la mise au point de programmes en permettant de visualiser plus longuement un écran fugitif.

La Figure 4.13 montre le début de l'affichage généré par le déroulement du programme de la Figure 4.12. Le fichier source, ADRESSES.LST, est le fichier BASIC de la Figure 4.10.

Si l'on compare le programme initial et le programme modifié (Figure 4.11 et Figure 4.12) ainsi que leurs résultats, il est aisé de comprendre les techniques de manipulation de chaînes du Turbo.

Quelques autres procédures et fonctions standard sont utiles pour la conversion de chaînes en autres types de données. Le Turbo interdit l'entrée directe à partir du clavier de données de type booléen ou défini par l'utilisateur. Par conséquent, la plupart des opérations d'entrée au clavier prennent des données numériques ou de chaîne et les convertissent en une autre forme. Nous avons tous utilisé des tests tels que ceux de la Figure 4.14 pour assigner une valeur à une variable. Dans cet exemple, on remarque que des *chaînes* sont assignées à la variable de chaîne Ticket.

La structure CASE de la Figure 4.15 est une autre méthode permettant de réaliser des opérations d'entrée à partir du clavier. Elle semble avoir la préférence des utilisateurs pour l'implémentation de menus.

```
PROCEDURE TraitTampon;
BEGIN
   LongTamp := 2;
   Index := 1;
   WHILE LongTamp > 1 DO
   BEGIN
      PremGuill := POS ('"', Tampon);
         WRITELN ('PremGuill = ', PremGuill);
      LongTamp := LENGTH (Tampon);
         WRITELN ('longTamp = ', LongTamp);
      Reste := COPY (Tampon, PremGuill + 1, LongTamp);
         WRITELN ('Reste = ', Reste);
      GuillSuiv := POS ('"', Reste);
         WRITELN ('GuillSuiv = ', GuillSuiv);
      Information := COPY (Reste, 1, GuillSuiv - 1);
      IF LENGTH (Information) <> 1 THEN
      BEGIN
         WRITELN;
         WRITELN ('Information = ', Information);
         DELAY (400);
         CASE Index OF
            1: Pers.Nom := Information;
            2: Pers.Prenom := Information;
            3: Pers.Rue := Information;
            4: Pers.Ville := Information;
            5: Pers.CodePost := Information;
          END;
          Index := Index + 1;
       END;
       Tampon := Reste;
    END;
    Tampon := '';
    WRITE (FichSortie, Pers);
END;
```

Figure 4.12 : Programme de la Figure 4.11 modifié pour afficher les résultats des procédures de manipulation de chaînes.

```
Nom du fichier BASIC : ADRESSES.LST
Nom du fichier Turbo :
ADRESSES.TBO
PremGuill = 1
longTamp = 64
Reste = Michut","Melle Adrienne","67, rue Ratounet","Les Arcs","73124"
GuillSuiv = 7

Information = Michut
PremGuill = 7
longTamp = 63
Reste = ,"Melle Adrienne","67, rue Ratounet","Les Arcs","73124"
GuillSuiv = 2
PremGuill = 2
longTamp = 56
Reste = Melle Adrienne","67, rue Ratounet","Les Arcs","73124"
GuillSuiv = 15

Information = Melle Adrienne
PremGuill = 15
longTamp = 54
Reste = ,"67, rue Ratounet","Les Arcs","73124"
GuillSuiv = 2
PremGuill = 2
longTamp = 39
Reste = 67, rue Ratounet","Les Arcs","73124"
GuillSuiv = 17

Information = 67, rue Ratounet
PremGuill = 17
longTamp = 37
Reste = ,"Les Arcs","73124"
GuillSuiv = 2
PremGuill = 2
longTamp = 20
Reste = Les Arcs","73124"
GuillSuiv = 9
```

Figure 4.13 : Résultat du déroulement du programme de la Figure 4.12.

```
Information = Les Arcs
PremGuill = 9
longTamp = 18
Reste = ,"73124"
GuillSuiv = 2
PremGuill = 2
longTamp = 9
Reste = 73124"
GuillSuiv = 6

Information = 73124
PremGuill = 6
longTamp = 7
Reste =
GuillSuiv = 0

Information =
PremGuill = 0
longTamp = 1
Reste =
GuillSuiv = 0

Information =
PremGuill = 2
longTamp = 64
Reste = Lemercier","M. Marcel","14, av des pré hauts","Paris","75016"
GuillSuiv = 10

Information = Lemercier
PremGuill = 10
longTamp = 62
Reste = ,"M. Marcel","14, av des pré hauts","Paris","75016"
GuillSuiv = 2
PremGuill = 2
longTamp = 52
Reste = M. Marcel","14, av des pré hauts","Paris","75016"
GuillSuiv = 10
```

Figure 4.13 (suite)

```
Information = M. Marcel
PremGuill = 10
longTamp = 50
Reste = ,"14, av des pré hauts","Paris","75016"
GuillSuiv = 2
PremGuill = 2
longTamp = 40
Reste = 14, av des pré hauts","Paris","75016"
GuillSuiv = 21

Information = 14, av des pré hauts
PremGuill = 21
longTamp = 38
Reste = ,"Paris","75016"
GuillSuiv = 2
PremGuill = 2
longTamp = 17
Reste = Paris","75016"
GuillSuiv = 6

Information = Paris
PremGuill = 6
longTamp = 15


Reste = ,"75016"
GuillSuiv = 2
PremGuill = 2
longTamp = 9
Reste = 75016"
GuillSuiv = 6

Information = 75016
PremGuill = 6
longTamp = 7
Reste =
GuillSuiv = 0
```

Figure 4.13 (suite)

```
Information =
PremGuill = 0
longTamp = 1
Reste =
GuillSuiv = 0


Information =
PremGuill = 2
longTamp = 65
Reste = Brécat","M. Jacques","116, impasse des fleurs","Orsay","91160"
GuillSuiv = 7
```

Figure 4.13 (suite)

```
{Ticket a été déclaré auparavant par STRING[10]}

   WRITELN ('Niveau : (N)Néophyte, (T)Technicien, (M)Moyen, (C)Chevronné, (S)Supérieur');
   READLN (Classe);
   IF (Classe = 'N') OR (Classe = 'n') THEN Ticket := 'Néophyte';
   IF (Classe = 'T') OR (Classe = 't') THEN Ticket := 'Technicien';
   IF (Classe = 'M') OR (Classe = 'm') THEN Ticket := 'Moyen';
   IF (Classe = 'C') OR (Classe = 'c') THEN Ticket := 'Chevronné';
   IF (Classe = 'S') OR (Classe = 's') THEN Ticket := 'Supérieur';
```

Figure 4.14 : Partie de programme illustrant l'assignation de chaînes à une variable de chaîne à l'aide d'instructions multiples IF/THEN.

Une autre technique dont on parle peu simule la fonction INKEY$ du BASIC dans laquelle une touche tapée est lue puis exécutée sans la frappe de la touche Return. Elle utilise le fichier d'entrée prédéclaré KBD. La Figure 4.16 en est une illustration ; cette technique n'est valable que pour l'entrée d'une seule touche.

```
{Option a été déclaré comme un entier}

   READLN (Option);
   CASE Option OF
      1: Accepte_Nouv_Noms;
      2: Cherche_Un_Nom;
      3: Liste_Alphabetique;
      4: Affiche_Liste_Sur_Ecran;
      5: Genere_Etiquettes;
      6: Cree_List_Adresses;
      7: Lit_Repertoire;
      8: Sortie;
   ELSE Gestion_erreur;
```

Figure 4.15 : Fragment d'un programme assignant des chaînes à une variable de chaîne à l'aide d'une structure CASE.

```
{Option a été déclaré comme un entier}

   READ (KBD, Option);
   CASE Option OF
      1: Accepte_Nouv_Noms;
      2: Cherche_Un_Nom;
      3: Liste_Alphabetique;
      4: Affiche_Liste_Sur_Ecran;
      5: Genere_Etiquettes;
      6: Cree_List_Adresses;
      7: Lit_Repertoire;
      8: Sortie;
   ELSE Gestion_erreur;
```

Figure 4.16 : Fragment de programme illustrant la simulation Turbo d'une fonction INKEY$ BASIC.

TYPES DE DONNÉES DÉFINIS PAR L'UTILISATEUR

On a souvent besoin de remplir des formulaires dans lesquels on entre une date de naissance sous la forme JJ-MM-AA qui représente le jour, le mois et l'année ou un code à deux chiffres pour la couleur des yeux, la couleur des cheveux, la taille ou la nationalité. Cette forme est tellement habituelle que l'on a presque oublié que les dates de naissance peuvent être exprimées de la manière suivante : 27 Janvier 1952 et que la couleur des yeux peut être caractérisée par les mots "bleus " ou "verts ". Le Turbo permet à l'utilisateur de créer ses propres types de données en utilisant des mots du langage courant dans n'importe quelle séquence reflétant la réalité.

Cela est très pratique, mais il ne faut pas oublier que le Turbo n'autorise pas les opérations d'entrée/sortie sur des types de données définis par l'utilisateur. Il est impossible par exemple de définir un type de données appelé Mois et d'entrer simplement le mot "Mai ". En Turbo comme en BASIC, on peut taper Mai sur l'écran mais le programme a besoin d'une procédure pour analyser la chaîne tapée et l'assigner à une variable du type Mois. La méthode du Turbo n'a donc aucun avantage par rapport à un programme BASIC dans lequel on tape le mois avec un sous-programme qui analyse l'entrée comme un nombre.

Cependant, alors que les types scalaires énumérés déclarés par l'utilisateur ne favorisent pas l'entrée, ils constituent un outil intéressant pour rechercher des bits de données dans des structures plus importantes. Ils facilitent certainement la lecture et la compréhension des programmes. Le programme ClubInformatique qui apparaît dans les pages suivantes contient le type de données appelé Grade. Comme n'importe quel type de données, il est déclaré au début du programme. Le fragment de programme de la Figure 4.17 illustre l'emploi d'un type de données créé par l'utilisateur. Le programme complet apparaît dans la Figure 4.19.

Nous avons défini un type de données appelé Grade. De plus, nous avons énuméré les éléments de ce type de données (Néophyte, Technicien, Moyen, Chevronné, Supérieur). Le Turbo doit connaître les éléments constituant un type de données, aussi bien défini par l'utilisateur que prédéfini. Les programmeurs qui ont conçu le compilateur Turbo ont dû également énumérer les élé-

```
{Attention, ce programme n'est pas complet !}

PROGRAM Demo_Enregistrement;

TYPE
   Grade = (Neophyte, Technicien, Moyen, Chevronne, Superieur);
   EnregMembre = RECORD
      Cle : STRING[30];
      AppelSigne : STRING[8];
      A_Jour : BOOLEAN;
      Doit_T : REAL;
      Niveau : Grade;
   END;

VAR
   Nom : STRING[30];
   Appel : STRING[8];
   Paye : BOOLEAN;
   Doit : REAL;
   Classe : CHAR;
   Ticket : Grade;
   Membre : EnregMembre;
.
.
.
BEGIN
.
.
.
   WRITELN ('Niveau : (N)Néophyte, (T)Technicien, (M)Moyen, (C)Chevronné, (S)Supérieur');
   READLN (Classe);
   IF (Classe = 'N') OR (Classe = 'n') THEN Ticket := 'Néophyte';
   IF (Classe = 'T') OR (Classe = 't') THEN Ticket := 'Technicien';
   IF (Classe = 'M') OR (Classe = 'm') THEN Ticket := 'Moyen';
   IF (Classe = 'C') OR (Classe = 'c') THEN Ticket := 'Chevronné';
   IF (Classe = 'S') OR (Classe = 's') THEN Ticket := 'Supérieur';
   Membre.Niveau := Ticket;
   WRITE (SortieFichier, Membre);
END;
```

Figure 4.17 : Fragment de programme qui illustre l'emploi d'un type de données créé par l'utilisateur.

ments des types de données prédéfinis. A l'intérieur de ce logiciel, le type de données BYTE (octet) a été défini comme comprenant les valeurs 0 à 255 et les données de type BOOLEAN (booléen), les valeurs TRUE et FALSE. On remarque qu'une fois le type défini, on peut déclarer comme lui appartenant toutes sortes d'identificateurs de variables. Niveau et Ticket sont des identificateurs du type Grade. Par conséquent, ils peuvent prendre les valeurs Néophyte, Technicien, Moyen, Chevronné ou Supérieur.

Dans cette partie de programme, le caractère ASCII tapé par l'opérateur en réponse à la première commande READLN est chargé dans la variable caractère Classe. Selon la valeur de classe, la variable Ticket prend l'une des valeurs possibles. A partir de là, Ticket peut être manipulé comme une variable scalaire quelconque. Les opérations suivantes sont toutes autorisées ; elles utilisent des variables telles que Ticket du type de données Grade :

```
FOR Ticket := Neophyte TO Chevronne DO...
IF (Ticket <Technicien) OR (Ticket >Chevronne) THEN...
WHILE Ticket <Moyen DO...
```

Les variables de type Grade peuvent même être triées ; l'ordre du tri est celui dans lequel elles ont été définies. Dans cet exemple, Néophyte précède Technicien qui précède Moyen, etc. Pour déterminer la position d'une variable à l'intérieur d'un type de données créé par l'utilisateur, le Turbo offre la fonction ORD. Elle est employée dans le programme au cours du processus de lecture des enregistrements. L'exemple de la Figure 4.18 combine la structure CASE à la fonction ORD. L'expression :

```
ORD (Niveau)
```

renvoie 0 si Niveau a la valeur Néophyte, 1 s'il a la valeur Technicien, etc. Ensuite, la structure CASE sélectionne la chaîne à afficher en fonction de l'entier fourni par la fonction ORD. Cette fonction a d'autres utilisations que la détermination de la position des types de données créés par l'utilisateur. Elle s'applique souvent à des caractères ASCII pour déterminer, par exemple, s'ils font partie des caractères alphanumériques (48-57, 65-90, 97-122).

Les types de données créés par l'utilisateur sont fréquemment employés dans les programmes Turbo sophistiqués. Comme nous l'avons mentionné précédemment, ils sont particulièrement appro-

```
PROCEDURE Lit_Fichier_Existant;
BEGIN
   READ (EntreeFichier, Membre); ...
   WITH Membre DO
   BEGIN
      CASE ORD (Niveau) OF
         0 : WRITELN ('Néophyte');
         1 : WRITELN ('Technicien');
         2 : WRITELN ('Moyen');
         3 : WRITELN ('Chevronné');
         4 : WRITELN ('Superieur');
      END;
END;
```

Figure 4.18 : Fragment de programme illustrant l'emploi de la fonction ORD avec un type de données créé par l'utilisateur.

priés à l'élaboration de jeux d'aventures dans lesquels des types de données tels que Monstres, Trésors, Pièces, Ogres, etc. sont à la mode. La plupart de ces types de données créent l' “environnement aventureux ” et sont employés par le programme de manière interne. Cependant, il n'est pas nécessaire de les utiliser pour les entrées/sorties d'écran directes, car la partie d'analyse syntaxique du programme convertit les réponses du joueur (toujours définies comme des chaînes) en une forme utilisable par le programme.

ENREGISTREMENTS

Les enregistrements font partie des structures de données les plus souples du Turbo. Les tableaux d'enregistrements peuvent être manipulés en mémoire pour un traitement très rapide et les fichiers d'enregistrements importants sont facilement stockés, lus ou écrits sur disque. Cette souplesse est déjà très intéressante, mais les enregistrements offrent un avantage encore plus grand : ils permettent de manipuler une collection d'éléments de données différents regroupés comme s'ils formaient un seul ensemble.

Par contre, les éléments d'un tableau doivent faire partie du même type. Dans un tableau de chaînes, chaque élément est une chaîne ; dans un tableau d'entiers, chaque élément est un entier. Cela signifie que pour "suivre " une combinaison Nom/Prix, il faut stocker tous les nombres sous la forme de chaînes et les convertir en valeurs numériques quand cela est nécessaire pour les calculs.

Un enregistrement Turbo reflète la structure de nombreux enregistrements de gestion standard. Un bordereau de vente ordinaire inclut normalement un nom, une adresse, un numéro de téléphone, une date, un nom de produit, une quantité, un prix, etc. Pour conserver un enregistrement de tels bordereaux, on a besoin d'une structure de données capable de rassembler les divers types et éléments dans une unité équivalente au bordereau de vente ; cette unité est l'enregistrement.

Le programme de la Figure 4.19 a été développé pour générer une liste des membres d'un club d'informatique. Le programme complet était un peu plus complexe car il incluait les adresses, les numéros de téléphone et d'autres informations. Ces éléments ont été supprimés pour réduire la liste et la rendre plus facile à manipuler. Néanmoins, le programme emploie assez peu de structures de données associées et de nouvelles procédures et illustre la capacité du Turbo à résoudre des problèmes de gestion courants avec efficacité et élégance.

On voit que le programme de la Figure 4.19 utilise à la fois un tableau d'enregistrements (ClubInformatique) et deux fichiers d'enregistrements (EntréeFichier et SortieFichier). De plus, il emploie un type de données énumérées créé par l'utilisateur Grade et convertit les entrées/sorties autorisées (caractères, entiers et nombres réels) en des données booléennes créées par l'utilisateur, qui peuvent être manipulées de façon interne mais ne peuvent pas servir pour des entrées/sorties directes. Il utilise également la fonction VAL. Chacune de ces caractéristiques est abordée dans ce chapitre.

Le programme sollicite des informations relatives aux membres du club et crée pour chaque membre un enregistrement appelé EnregMembre, composé des divers éléments qui le décrivent. Nous avons choisi un exemple qui n'emploie pas seulement des caractères et des entiers mais également un type de données créé par l'utilisateur (Grade qui représente le niveau), ainsi que des nombres réels correspondant à une somme et surtout une valeur booléenne (AJour qui caractérise un état oui/non). La capacité d'entrer un élément de données booléen dans un enregistrement est une pos-

```
PROGRAM Demo_Enregistrement;

TYPE
   Grade = (Neophyte, Technicien, Moyen, Chevronne, Superieur);

   EnregMembre = RECORD
      Cle : STRING[30];
      AppelSigne : STRING[8];
      A_Jour : BOOLEAN;
      Doit_T : REAL;
      Niveau : Grade;
   END;

VAR
   Nom : STRING[30];
   Appel : STRING[8];
   Paye : BOOLEAN;
   Doit : REAL;
   Classe : CHAR;
   Ticket : Grade;
   Membre : EnregMembre;
   ClubInformatique : ARRAY[1..73] OF EnregMembre;
   EntreeFichier, SortieFichier : FILE OF EnregMembre;
   Reponse : STRING[30];
   ToutFait : BOOLEAN;

PROCEDURE Charge_Tableau;
VAR
   Montant : STRING[6];
   Resultat : INTEGER;

BEGIN
   ToutFait := FALSE;
   WRITELN ('Nom du fichier contenant les noms :');
   READLN (Reponse);
   ASSIGN (SortieFichier, Reponse);
   REWRITE (SortieFichier);
   WHILE NOT ToutFait DO
   BEGIN
      WRITELN ('Nom :');
      READLN (Nom);
      Membre.Cle := Nom;
      WRITELN ('Appel :');
      READLN (Appel);
      Membre.AppelSigne := Appel;
      WRITELN ('Combien doit-il ?');
      READLN (Montant);
```

Figure 4.19 : Programme illustrant l'emploi d'enregistrements.

```
      VAL (Montant, Doit, Resultat);
      IF Doit < 0.01 THEN Paye := TRUE ELSE Paye := FALSE;
      Membre.A_Jour := Paye;
      Membre.Doit_T := Doit;
      WRITELN ('Niveau : (N)Néophyte, (T)Technicien, (M)Moyen, (C)Chevronné, (S)Supérieur');
      READLN (Classe);
      IF (Classe = 'N') OR (Classe = 'n') THEN Ticket := Neophyte;
      IF (Classe = 'T') OR (Classe = 't') THEN Ticket := Technicien;
      IF (Classe = 'M') OR (Classe = 'm') THEN Ticket := Moyen;
      IF (Classe = 'C') OR (Classe = 'c') THEN Ticket := Chevronne;
      IF (Classe = 'S') OR (Classe = 's') THEN Ticket := Superieur;
      Membre.Niveau := Ticket;
      WRITE (SortieFichier, Membre);
      WRITELN ('Taper un astérisque pour terminer ; sinon Return');
      READLN (Reponse);
      IF Reponse = '*' THEN ToutFait := TRUE;
   END;
   CLOSE (SortieFichier);
END;

PROCEDURE Lit_Fichier_Existant;
BEGIN
   WRITELN ('Nom du fichier contenant les enregistrements du club :');
   READLN (Reponse);
   ASSIGN (EntreeFichier, Reponse);
   RESET (EntreeFichier);
   WHILE NOT EOF (EntreeFichier) DO
   BEGIN
      READ (EntreeFichier, Membre);
      WITH Membre DO
      BEGIN
         WRITE (Cle,' - - - ');
         WRITELN (AppelSigne);
         CASE ORD (Niveau) OF
            0 : WRITELN ('Néophyte');
            1 : WRITELN ('Technicien');
            2 : WRITELN ('Moyen');
            3 : WRITELN ('Chevronné');
            4 : WRITELN ('Superieur');
         END;
      END;
   END;
   CLOSE (EntreeFichier);
END;

BEGIN
   WRITELN;
   WRITELN ('Voulez-vous afficher l ancien fichier ?');
   READLN (Reponse);
   IF Reponse = 'O' THEN Lit_Fichier_Existant ELSE Charge_Tableau;
END.
```

Figure 4.19 (suite)

sibilité puissante pour représenter un état, actif/inactif, membre/non-membre.

Comme toutes les variables du Turbo, un enregistrement doit être déclaré avec un identificateur et un type de données. Il faut également déclarer les éléments de cet enregistrement. On remarque la syntaxe car elle est inhabituelle. La déclaration commence par le mot réservé RECORD et se termine par le mot réservé END. Les enregistrements peuvent inclure parmi leurs éléments d'autres enregistrements.

De nombreux programmeurs Pascal font la confusion entre la déclaration d'un type de données et la déclaration d'un type de variables. D'une manière ou d'une autre, ce genre de confusion émerge lorsqu'ils manipulent des enregistrements. Personne n'aurait l'idée d'écrire des instructions telles que :

```
REAL := 0.05 ;
CHAR := 'A' ;
```

en utilisant les mots réservés REAL et CHAR comme s'il s'agissait de variables, alors qu'il faut taper :

```
TauxTaxes := 0.05 ;
GradeSup := 'A' ;
```

Dans ce dernier exemple, les identificateurs TauxTaxe et GradeSup doivent avoir été déclarés auparavant de type REAL et CHAR. Cependant, une erreur courante consiste à penser qu'un enregistrement particulier est une variable et non un type de données. Dans l'exemple de programme, EnregMembre est un type de données et non une variable ; par conséquent, il est impossible de lui assigner des valeurs. D'autre part, la déclaration de variables contient les lignes suivantes :

```
VAR
    Membre : EnregMembre ;
    ClubInformatique : ARRAY [1..73] OF EnregMembre ;
    EntreeFichier, SortieFichier : FILE OF EnregMembre ;
```

Membre est un identificateur de variables auquel on doit assigner des valeurs. Il est conseillé d'employer des noms similaires pour un type d'enregistrement et pour les variables qui lui sont associées

(Membre et EnregMembre dans cet exemple). On observe la déclaration d'enregistrement :

```
TYPE
      EnregMembre = RECORD
            Cle : STRING [30] ;
            AppelSigne : STRING [8] ;
            AJour : BOOLEAN ;
            Doit : REAL ;
            Niveau : Grade ;
      END ;
```

A l'intérieur de l'enregistrement, chaque élément de données a son propre identificateur. Dans la variable Membre (mais pas dans le type de données EnregMembre), les éléments sont :

```
Membre.Cle
Membre.AppelSigne
Membre.AJour
Membre.Doit
Membre.Niveau
```

Ensemble, ces éléments constituent la variable d'enregistrement Membre. Pour reprendre le point d'introduction, les enregistrements permettent de manipuler comme une seule entité un groupe d'éléments de types différents associés. On peut accéder à des enregistrements, les lire, les écrire, les trier et les placer sous forme de tableaux comme s'il s'agissait de types de données simples (caractères par exemple) tout en pouvant toujours accéder à une partie d'un enregistrement quelconque.

L'exemple de programme sollicite des informations de l'utilisateur pour créer un fichier. Bien sûr, une fois créé le fichier de données peut être manipulé, effacé, ajouté ou trié par un programme sans interactions avec le clavier. Des exemples ultérieurs illustreront ces opérations.

La méthode utilisée par le programme pour signaler la fin de l'entrée à l'aide d'un astérisque (*) n'est pas très élégante ; il en existe d'autres.

Le programme de la Figure 4.19 demande des informations relatives à un membre du club. Ensuite, il convertit la chaîne et les nombres réels entrés au clavier en des données définies par l'uti-

lisateur et stocke les réponses individuelles dans un groupe de variables. Voici la liste de ces variables et de leur type :

Nom de variable	**Type de données**
Nom	STRING [30]
Appel	STRING [8]
Paye	BOOLEAN
Doit	REAL
Classe	CHAR
Ticket	Grade

Ces variables sont ensuite assignées à leurs éléments respectifs pour chaque enregistrement de membre :

Élément d'enregistrement	**Type de données**
Membre.Cle	STRING [30] ;
Membre.AppelSigne	STRING [8] ;
Membre.AJour	BOOLEAN ;
Membre.Doit	REAL ;
Membre.Niveau	Grade ;

Les déclarations de types apparaissent ici pour deux raisons : souligner la possibilité pour les enregistrements de stocker des éléments de données différents, et montrer à l'utilisateur que, pour chaque élément d'une déclaration d'enregistrement, on a généralement besoin d'une variable de même type pour contenir des informations écrites, lues ou assignées à l'élément d'enregistrement.

Les différents niveaux ont des positions très précises (Néophyte a pour rang 0, Technicien 1, etc.) utilisées dans l'instruction CASE. Ces positions autorisent également les recherches et les tests. L'utilisateur peut réfléchir à l'exploitation de cette capacité dans ses propres applications. La fonction standard ORD renvoie la position d'une variable scalaire quelconque (excepté bien sûr d'un nombre réel). Elle est utilisée plusieurs fois dans ce livre. Il ne faut pas oublier, dans ce cas, que le premier élément a toujours la position 0 et non 1.

Une fois les valeurs assignées à tous les éléments d'un enregistrement, on emploie généralement une forme d'indexation pour accéder à l'enregistrement suivant. Dans l'exemple de programme, chaque enregistrement complet est écrit dans le fichier avec la commande WRITE.

WRITE (SortieFichier, Membre) ;

Après l'écriture d'un enregistrement, on peut choisir d'entrer l'enregistrement suivant ou d'arrêter l'entrée et de fermer le fichier. La Figure 4.20 montre une partie de session interactive acceptant les entrées.

```
Voulez-vous afficher l ancien fichier ?
N
Nom du fichier contenant les noms :
INFOCLUB.LST
Nom :
Alphonse Michut
Appel :
WN2MYU
Combien doit-il ?
350
Niveau : (N)Néophyte, (T)Technicien, (M)Moyen, (C)Chevronné, (S)Supérieur
N
Taper un astérisque pour terminer ; sinon Return

Nom :
Anne Baudet
Appel :
WB2ISA
Combien doit-il ?
720
Niveau : (N)Néophyte, (T)Technicien, (M)Moyen, (C)Chevronné, (S)Supérieur
T
Taper un astérisque pour terminer ; sinon Return
*
```

Figure 4.20 : Déroulement de l'option de création de fichier du programme de la Figure 4.19.

La deuxième option du programme de démonstration lit un fichier de données existant puis le traite. Dans cet exemple, le programme affiche le fichier sur l'écran. Des enregistrements complets sont lus puis manipulés ; la manipulation inclut la conversion des types de données créés par l'utilisateur en une forme plus lisible. D'autres tâches auraient pu inclure l'addition des sommes dues au club ou le tri de la liste par nom ou par niveau. La Figure 4.21 montre une partie de l'affichage lorsque l'option d'affichage est sélectionnée.

```
Voulez-vous afficher l ancien fichier ?
O
Nom du fichier contenant les enregistrements du club :
INFOCLUB.LST
Alphonse Michut - - - WN2MYU
Néophyte
Anne Baudet - - - WB2ISA
Technicien
```

Figure 4.21 : Déroulement de l'option d'affichage de fichier du programme de la Figure 4.19.

Ce programme est un exemple particulièrement intéressant de l'emploi de structures de données parce qu'il combine les enregistrements avec des tableaux ou des fichiers. Au lieu de se contenter de lire les enregistrements, il aurait pu les charger dans un tableau pour les trier. L'annexe de cet ouvrage propose un programme de mailing qui applique la méthode de tri de Shell de la Figure 4.1 au tri d'un tableau d'enregistrements de clients. Ce programme peut trier (ordonner) des éléments alphabétiquement ou par code postal.

Si les enregistrements ordinaires ne sont pas assez souples, le Turbo offre une structure de données encore plus souple appelée *enregistrement avec variantes*. Ces enregistrements ne diffèrent pas seulement des autres par les données stockées dans chaque élément mais peuvent avoir différentes combinaisons d'éléments de données. Ils forment une structure particulièrement utile pour organiser des données de manière intelligible pour les utilisateurs.

Malheureusement, cette caractéristique est très peu documentée ; même les ouvrages relatifs au Pascal semblent ignorer cette capacité, peut-être parce que la seule manière d'illustrer son fonctionnement est de développer des programmes très sophistiqués. C'est exactement ce que fait le programme de la Figure 4.22.

Le programme de la Figure 4.22 montre le programme ClubInformatique modifié pour utiliser des enregistrements avec variantes. Il garde maintenant une trace des intérêts des différents membres. Les membres sont séparés en opérateurs et expérimentateurs. Une fois cette distinction réalisée, le fichier contient deux enregistrements différents, chacun d'eux étant composé de divers éléments d'information.

Les enregistrements avec variantes se prêtent aux programmes de gestion dans lesquels des informations sont communes à tous les clients, mais on peut également enregistrer des informations supplémentaires, différentes pour chaque type de client. Une autre application peut consister à garder une trace des employés d'une entreprise mensualisés, à salaire horaire ou employés de façon temporaire. Ils ont tous un nom, une adresse et un numéro de Sécurité sociale, mais le type de salaire, le temps de travail et les horaires ne sont pas identiques pour tous. Les capacités des enregistrements avec variantes sont uniquement limitées par le degré d'imagination et d'ingéniosité du programmeur.

Ce type d'enregistrement peut être associé à un type de données quelconque. Dans notre exemple, les enregistrements avec variantes des expérimentateurs contiennent des éléments booléens mais ceux des opérateurs incluent une valeur entière et une valeur booléenne. On aurait pu employer n'importe quel type de données (caractères, nombre réels, etc.) et même avoir des enregistrements à l'intérieur des enregistrements. Bien que cette version soit plus complexe que la précédente, elle offre de plus grandes possibilités. Par exemple, elle permet de générer des rapports sélectifs concernant uniquement les expérimentateurs ou uniquement les opérateurs.

Ce programme illustre un "raccourci " de programmation utile avec les enregistrements, la construction WITH. Il s'agit simplement d'un moyen pratique pour limiter la frappe lors du développement de programmes sans affecter sa vitesse d'exécution. Bien que les enregistrements soient en principe manipulés comme une unité, toutes les implémentations Pascal offrent la possibilité de fragmenter un enregistrement pour utiliser les données stockées dans

```
PROGRAM Demo_Enregistrement_Variant;

TYPE
   Grade = (Neophyte, Technicien, Moyen, Chevronne, Superieur);
   InteretPrimaire = (Operateur, Experimentateur);

   EnregMembre = RECORD
      Cle : STRING[30];
      AppelSigne : STRING[8];
      A_Jour : BOOLEAN;
      Doit_T : REAL;
      Niveau : Grade;
      TypeOp : InteretPrimaire;
      CASE OpInteret : InteretPrimaire OF {placé à la fin des déclarations}
         Operateur :
            (NuVal : INTEGER;          {l'usage des parenthèses est crucial}
            Trafic : BOOLEAN);
         Experimentateur :
            (VHF : BOOLEAN;
            Ordinateur : BOOLEAN);
   END;

VAR
   Nom : STRING[30];
   Appel : STRING[8];
   Paye : BOOLEAN;
   Doit : REAL;
   Classe : CHAR;
   Ticket : Grade;
   Membre : EnregMembre;
   ClubInformatique : ARRAY[1..73] OF EnregMembre;
   EntreeFichier, SortieFichier : FILE OF EnregMembre;
   Reponse : STRING[30];
   ToutFait : BOOLEAN;
   OpInteret : InteretPrimaire;

PROCEDURE Charge_Tableau;
VAR
   Montant : STRING[6];
   Index, Resultat, Valeur : INTEGER;
   InteretEO, TraficON, VHFON, OrdinateurON : CHAR;

BEGIN
   ToutFait := FALSE;
   Index := 1;
   WRITELN ('Nom du fichier contenant les noms :');
   READLN (Reponse);
```

Figure 4.22 : Programme de la Figure 4.19 modifié pour illustrer l'utilisation d'enregistrements avec variantes.

```
ASSIGN (SortieFichier, Reponse);
REWRITE (SortieFichier);
WHILE NOT ToutFait DO
BEGIN
   WRITELN ('Nom :');
   READLN (Nom);
   Membre.Cle := Nom;
   WRITELN ('Appel :');
   READLN (Appel);
   Membre.AppelSigne := Appel;
   WRITELN ('Combien doit-il ?');
   READLN (Montant);
   VAL (Montant, Doit, Resultat);
   IF Doit < 0.01 THEN Paye := TRUE ELSE Paye := FALSE;
   Membre.A_Jour := Paye;
   Membre.Doit_T := Doit;
   WRITELN ('Niveau : (N)Néophyte, (T)Technicien, (M)Moyen, (C)Chevronné, (S)Supérieur');
   READLN (Classe);
   IF (Classe = 'N') OR (Classe = 'n') THEN Ticket := Neophyte;
   IF (Classe = 'T') OR (Classe = 't') THEN Ticket := Technicien;
   IF (Classe = 'M') OR (Classe = 'm') THEN Ticket := Moyen;
   IF (Classe = 'C') OR (Classe = 'c') THEN Ticket := Chevronne;
   IF (Classe = 'S') OR (Classe = 's') THEN Ticket := Superieur;
   Membre.Niveau := Ticket;
   WRITELN ('Est-ce un (E)expérimentateur ou un (O)pérateur ?');
   READLN (InteretEO);
   IF (InteretEO = 'E') OR (InteretEO = 'e') THEN OpInteret := Experimentateur;
   IF (InteretEO = 'O') OR (InteretEO = 'o') THEN OpInteret := Operateur;
   Membre.TypeOp := OpInteret;
   IF OpInteret = Operateur THEN
   BEGIN
      WRITELN ('Sur combien de bandes travaille-t-il ?');
      READLN (Valeur);
      Membre.NuVal := Valeur;
      WRITELN ('Collabore-t-il à un réseau ? (O/N)');
      READLN (TraficON);
      IF (TraficON = 'O') OR (TraficON = 'o') THEN Membre.Trafic := TRUE
      ELSE Membre.Trafic := FALSE;
   END;
   IF OpInteret = Experimentateur THEN
   BEGIN
      WRITELN ('Est-il intéressé par la VHF ? (O/N)');
      READLN (VHFON);
      VHFON := UPCASE (VHFON);
      IF VHFON = 'O' THEN Membre.VHF :=TRUE ELSE Membre.VHF := FALSE;
      WRITELN ('Est-il intéressé par les ordinateurs ? (O/N)');
      READLN (OrdinateurON);
      OrdinateurON := UPCASE(OrdinateurON);
      IF OrdinateurON = 'O' THEN Membre.Ordinateur := TRUE
      ELSE Membre.Ordinateur := FALSE;
   END;
   WRITE (SortieFichier, Membre);
```

Figure 4.22 (suite)

```
        Index := Index + 1;
        WRITELN ('Taper un astérisque pour terminer ; sinon Return');
        READLN (Reponse);
        IF Reponse = '*' THEN ToutFait := TRUE;
    END;
    CLOSE (SortieFichier);
END;

PROCEDURE Lit_Fichier_Existant;
BEGIN
    WRITELN ('Nom du fichier contenant les enregistrements du club :');
    READLN (Reponse);
    ASSIGN (EntreeFichier, Reponse);
    RESET (EntreeFichier);
    WHILE NOT EOF (EntreeFichier) DO
    BEGIN
        READ (EntreeFichier, Membre);
        WITH Membre DO
        BEGIN
            WRITE (Cle,' · · · ');
            WRITELN (AppelSigne);
            CASE ORD (Niveau) OF
                0 : WRITELN ('Néophyte');
                1 : WRITELN ('Technicien');
                2 : WRITELN ('Moyen');
                3 : WRITELN ('Chevronné');
                4 : WRITELN ('Superieur');
            END;
            IF A_Jour THEN WRITELN ('Membre à jour')
            ELSE WRITELN ('Doit : ',Doit :6:2);
            IF TypeOP = Operateur THEN
            BEGIN
                WRITELN ('Intéressé principalement par l électronique');
                WRITELN ('et par l émission sur la bande : ', NuVal);
                IF Trafic THEN WRITELN ('Intéressé par l élaboration d un réseau')
                ELSE WRITELN ('Pas intéressé par les réseaux');
            END;
            IF TypeOP = Experimentateur THEN
            BEGIN
                WRITELN ('Intéret principal pour l expérimentation');
                IF VHF THEN WRITELN ('et pour la VHF')
                ELSE WRITELN ('et pour la haute fréquence');
                IF Ordinateur THEN WRITELN ('ainsi que pour les ordinateurs')
                ELSE WRITELN ('mais pas par les ordinateurs');
            END;
            WRITELN;
        END;
    END;
    CLOSE (EntreeFichier);
END;

BEGIN
```

Figure 4.22 (suite)

```
    WRITELN;
    WRITELN ('Voulez-vous afficher l ancien fichier ?');
    READLN (Reponse);
    IF (Reponse = 'O') OR (Reponse = 'o') THEN Lit_Fichier_Existant
    ELSE Charge_Tableau;
END.
```

Figure 4.22 (suite)

les éléments qui le constituent. Chaque élément est identifié par la combinaison du nom de l'enregistrement et du nom de l'élément de donnée. Dans l'exemple ci-dessous, les éléments sont Membre.Clé, Membre.AppelSigne et Membre.Niveau :

```
WRITE (Membre.Cle, '---') ;
    WRITELN (Membre.AppelSigne) ;
    CASE ORD (Membre.Niveau) OF
        0 : WRITELN ('Neophyte') ;
.
.
.
```

Avec WITH, il n'est pas nécessaire de taper l'identificateur d'enregistrement (Membre) chaque fois. A la place, on commence le bloc de code par une ligne qui indique au Turbo qu'entre les mots réservés WITH et END toutes les variables font partie de l'enregistrement Membre. Cette méthode est illustrée par le fragment de programme suivant :

```
WITH Membre DO
    BEGIN
        WRITE (Cle, '---') ;
        WRITELN (.AppelSigne) ;
        CASE ORD (.Niveau) OF
            0 : WRITELN ('Neophyte') ;
.
.
.
END ;
```

Le Turbo permet de demander l'entrée de mots, d'entiers ou de simples réponses oui/non. Encore plus important, le programme est facile à développer et à mettre au point pour le programmeur. Celui-ci ne doit pas constamment garder une trace des codes arbitraires et de ce qu'ils représentent.

On aurait pu suivre les niveaux des membres dans n'importe quel langage de programmation avec des nombres 0, 1, etc., mais il est plus facile pour un programmeur de travailler sur un type de données appelé Grade avec des valeurs scalaires appelées Néophyte, Technicien, etc. Il n'est guère étonnant que le Turbo intéresse de plus en plus le développement de logiciels commerciaux.

Les mécanismes d'implémentation des enregistrements avec variantes demandent une certaine habitude. Il faut établir un type de données et déclarer une variable de ce type que le Turbo peut employer pour sélectionner l'enregistrement approprié. Dans notre exemple, nous avons créé un type de données appelé IntérêtPrimaire, qui a été défini comme se composant des éléments opérateurs et expérimentateurs.

Ensuite, nous avons déclaré une variable appelée OpIntérêt du type IntérêtPrimaire. Lors du déroulement du programme, la variable OpIntérêt contient la valeur qui indique si le membre du club est un opérateur ou un expérimentateur. Ensuite, la structure CASE (associée aux enregistrements avec variantes) choisit la variante d'enregistrement en fonction de la valeur de la variable OpIntérêt et les valeurs sont ajoutées à l'enregistrement de base commun à tous les membres du club.

La ponctuation et la syntaxe employées pour déclarer des enregistrements avec variantes sont très importantes (et peu abordées dans ce manuel). Tous les éléments d'enregistrement connectés avec chacune des options CASE sont placés entre parenthèses. Même la syntaxe de l'instruction CASE n'est pas une construction simple du type "CASE NomVariable OF " ; elle emploie une combinaison d'un nom de variable et d'un type de données :

```
TYPE
        InteretPrimaire = (Operateur, Experimentateur) ;

        EnregMembre = RECORD
                Cle : STRING [30] ;
                AppelSigne : STRING [8] ;
                TypeOp : InteretPrimaire ;
```

```
{Dans la ligne ci-dessous, InteretOp est une variable de type
InteretPrimaire. InteretOp contient les donnees sollicitees de
l'utilisateur et qui seront ensuite employees pour selectionner la
variante d'enregistrement qui doit etre utilisee pour
l'enregistrement de ce membre}

CASE InteretOp : InteretPrimaire OF
        Operateur :
                (Valeur : INTEGER ; {parentheses ouvrante}
                        Trafic : BOOLEAN) ; {... et fermante}
                Experimentateur :
                        (VHF : BOOLEAN ; {parentheses ouvrante}
                        Ordinateur : BOOLEAN) ; {... et fermante}
END ;
```

Une illustration plus générique suit. Il ne faut pas oublier que le nombre d'enregistrements avec variantes est quelconque, que la partie variante peut avoir des nombres d'éléments de données différents et que les éléments de données peuvent être constitués d'une combinaison de types de données quelconque.

```
TYPE
        Selection = (Choix1, Choix2, Choix3) ;
        TotalEnreg = RECORD
                EleStand1 : STRING [30] ; {Type de donnees
                                            quelconque}
                EleStand2 : STRING [8] ;
CASE TestVariable : Selection OF
        Choix1 :
                (EleOption1 : INTEGER ; {Type de donnees
                                         quelconque}
                EleOption2 : INTEGER ;
                EleOption3 : INTEGER ;)
        Choix2 :
                (EleOptionA : INTEGER ; {Type de donnees
                                         quelconque}
                EleOptionB : INTEGER ;)
        Choix3 :
                (EleOptionX : INTEGER ; {Type de donnees
                                         quelconque}
                EleOptionY : INTEGER ;)
END ;
```

```
VAR
      TestVariable : Selection ; {TestVariable est du type
                                  Selection}
```

Bien sûr, le Turbo affiche les messages d'erreur habituels si l'on oublie les parenthèses, mais la ponctuation est suffisamment peu courante pour que l'erreur soit difficile à détecter.

Dans la phase d'entrée du programme, on devait indiquer au Turbo la variété d'enregistrements désirée. L'exemple de programme obtient ses informations par l'intermédiaire de la question :

```
WRITELN ('Est-ce un (E)experimentateur ou un (O)operateur ?') ;
```

mais un test interne exécuterait aussi bien le travail. En fait, si l'on regarde la phase de sortie du programme, on s'aperçoit qu'un test interne (par opposition à un test dans lequel l'utilisateur effectue un choix interactif par l'intermédiaire du clavier) sélectionne l'information à afficher en évaluant le champ TypeOp de chaque enregistrement. Dans une application de gestion, ce genre de test peut choisir si une lettre de recouvrement doit être ou non générée. La Figure 4.23 montre le déroulement de la dernière version du programme sollicitant une nouvelle entrée et générant un nouveau fichier appelé MEMBRES.LST.

```
Voulez-vous afficher l ancien fichier ?
N
Nom du fichier contenant les noms :
MEMBRES.LST
Nom :
MARCEL DEBASE
Appel :
WA2YEO
Combien doit-il ?
535
Niveau : (N)Néophyte, (T)Technicien, (M)Moyen, (C)Chevronné, (S)Supérieur
T
Est-ce un (E)expérimentateur ou un (O)pérateur ?
Est-il intéressé par la VHF ? (O/N)
O
```

Figure 4.23 : Déroulement d'une opération de création de fichier à l'aide du programme de la Figure 4.22.

```
Est-il intéressé par les ordinateurs ? (O/N)
O
Taper un astérisque pour terminer ; sinon Return

Nom :
JULES WEBSTER
Appel :
WB2ISA
Combien doit-il ?

Niveau : (N)Néophyte, (T)Technicien, (M)Moyen, (C)Chevronné, (S)Supérieur
S
Est-ce un (E)expérimentateur ou un (O)pérateur ?
Sur combien de bandes travaille-t-il ?
5
Collabore-t-il à un réseau ? (O/N)
N
Taper un astérisque pour terminer ; sinon Return

Nom :
JACQUES ARRE
Appel :
WA1KWJ
Combien doit-il ?

Niveau : (N)Néophyte, (T)Technicien, (M)Moyen, (C)Chevronné, (S)Supérieur
C
Est-ce un (E)expérimentateur ou un (O)pérateur ?
Est-il intéressé par la VHF ? (O/N)
N
Est-il intéressé par les ordinateurs ? (O/N)
N
Taper un astérisque pour terminer ; sinon Return
*
```

Figure 4.23 (suite)

La Figure 4.24 montre le résultat d'un second déroulement du programme affichant le fichier MEMBRES.LST.

```
Voulez-vous afficher l ancien fichier ?
O
Nom du fichier contenant les enregistrements du club :
MEMBRES.LST
MARCEL DEBASE - - - WA2YEO
Technicien
Doit : 535.00
Intéret principal pour l expérimentation
et pour la VHF
ainsi que pour les ordinateurs

JULES WEBSTER - - - WB2ISA
Superieur
Membre à jour
Intéressé principalement par l électronique
et par l émission sur la bande : 5
Pas intéressé par les réseaux

JACQUES ARRE - - - WA1KWJ
Chevronné
Membre à jour
Intéret principal pour l expérimentation
et pour la haute fréquence
mais pas par les ordinateurs
```

Figure 4.24 : Affichage du fichier créé par le programme de la Figure 4.22.

Bien que la réponse à la question "Est-il intéressé par les ordinateurs ? " soit stockée comme une valeur booléenne (Figure 4.23), dans le résultat (Figure 4.24), la valeur n'est pas seulement employée pour afficher "oui " ou "non " mais aussi pour donner une réponse plus explicite. On aurait pu appeler une procédure pour exécuter presque tout ce qui est basé sur cette valeur booléenne simple.

Des ouvrages entiers pourraient traiter de l'utilisation d'enregistrements, mais les informations présentées ici sont suffisantes

pour l'écriture de programmes efficaces. Il est pratique de pouvoir stocker différents enregistrements à l'intérieur d'un fichier ou dans un tableau. Ces enregistrements peuvent caractériser différents sujets (clients, membres d'un club, etc.) et évitent (grâce à la partie variable) l'utilisation d'enregistrements très vastes constitués d'un grand nombre de champs qui sont inemployés.

Pour faire un travail de programmation sérieux avec le Turbo, il faut explorer les enregistrements standard et les enregistrements avec variantes. Il est conseillé de les étudier avec soin pour découvrir les possibilités qu'ils peuvent offrir dans différentes situations de programmation.

FICHIERS

Présentation

La plupart des programmeurs concèdent qu'une manipulation de fichiers bien menée garantit une bonne programmation. De nombreux programmes écrits pour des applications pratiques nécessitent la manipulation de fichiers externes. L'aptitude à écrire et à lire un fichier externe différencie le programmeur chevronné et le programmeur débutant.

Le Turbo offre une collection inégalée de capacités de manipulation de fichiers dépassant celles de n'importe quel autre Pascal. Les informaticiens remarquent que la manipulation de fichiers a toujours été l'une des faiblesses du Pascal standard. Une fois de plus, le Turbo pallie ces inconvénients en sacrifiant la compatibilité à une puissance étonnante. Étant donné ces innovations et ces extensions, les livres concernant le Pascal offrent peu d'aide, y compris pour le fichier d'entrée/sortie le plus élémentaire du Turbo. Pis encore, les messages d'erreur du Turbo, en ce qui concerne la manipulation de fichiers, ne sont pas aussi explicites que les autres.

Cette partie traite de la lecture et de l'écriture de tous les types de fichiers Turbo, y compris des plus complexes et des plus caractéristiques, les fichiers texte. Une fois leur gestion assimilée, on peut consulter d'autres ouvrages pour tout renseignement supplémentaire.

FICHIERS EXTERNES

Certains programmes doivent traiter des données externes. En effet, il y a une limite à la taille d'un programme : 64 K, sauf si l'on utilise des techniques particulières, chaînage ou recouvrement ; le Turbo permet également l'accès à 64 K de données. Par conséquent, chaque fois qu'un programme et ses données dépassent la taille autorisée, il faut employer des fichiers de données séparés. Le programme qui utilise une liste d'adresses, un répertoire téléphonique ou des enregistrements de clients nécessite le stockage des données dans un fichier séparé. Plus important encore, ces fichiers de données externes peuvent exister en dehors des opérations de création et de déroulement d'un programme ; par conséquent, un fichier de données peut être copié, archivé et même vendu comme une entité à part.

Par définition, les traitements de texte travaillent sur des fichiers externes. A l'inverse des programmes, les fichiers externes sont de simples collections de données utilisables par un programme Turbo et sont terminés par un indicateur de fin de fichier autorisé. Ces données peuvent être stockées sous la forme de texte, éventuellement avec des indicateurs de fin de ligne, ou comme une série d'enregistrements structurés de manière rigide. L'accès aux différents éléments d'un fichier est soit séquentiel, soit aléatoire (direct) ; les fichiers sont également dynamiques. Alors qu'il est nécessaire de spécifier les champs d'un enregistrement, le nombre d'éléments d'un tableau et le nombre de caractères d'une chaîne, les fichiers n'ont pas de taille prédéfinie. Cela signifie que l'on peut élaborer des fichiers aussi grands ou aussi petits que nécessaire.

Les fichiers ne sont pas limités par la mémoire de l'ordinateur mais par la manière dont certaines structures (telles que les tableaux) sont stockées. En théorie, avec un disque dur de 10 méga-octets, le fichier peut occuper la totalité du disque, même si la mémoire centrale de l'ordinateur ne dépasse pas 128 K. On peut écrire un programme d'application Turbo pour prendre en charge la gestion d'un fichier aussi monstrueux ; il ne pourra accéder simultanément qu'à des parties du fichier. La gestion des fichiers fait partie des notions les plus souples et les plus puissantes mises à la disposition du programmeur.

Note relative aux types de fichiers

Fichiers binaires et ASCII

Une fois créés par le Turbo, les fichiers sont gérés par le système d'exploitation, CP/M. Le système d'exploitation stocke et charge les fichiers, génère des répertoires, offre des facilités de copie, d'attribution de noms et de concaténation, et assure leur transfert entre les disques, la mémoire et les périphériques d'entrée/sortie (modem, imprimante, etc.). Lorsqu'un fichier est créé avec le Turbo Pascal, on peut employer toutes les capacités du système d'exploitation pour le manipuler.

Les utilisateurs qui ont essayé d'accéder à un fichier. COM ou.BIN savent que certains fichiers sont stockés sous une forme qui n'utilise pas le code ASCII et les conventions de fin de ligne attendues lors de l'ouverture d'un fichier texte ou d'un fichier BASIC ; par exemple, le système d'exploitation (DOS) utilise des fichiers binaires (.COM). Comme un système d'exploitation, le Turbo peut créer différents types de fichiers. On a déjà vu que l'éditeur Turbo emploie des fichiers texte ordinaires (.PAS) auxquels n'importe quel programme de traitement de texte peut accéder. On a également rencontré des fichiers binaires (.COM) créés lors de la compilation. Le Turbo a des fichiers logiques prédéfinis qui ont été employés dans les instructions READ et WRITE : INPUT, OUTPUT, LST, KBD, CON, etc. Ceux-ci ne sont pas de vrais fichiers qui apparaissent dans des répertoires.

Le Turbo est capable de créer des fichiers binaires, c'est-à-dire des fichiers composés de bits (chiffres binaires) ne pouvant pas être interprétés par le Turbo comme des caractères ASCII (.PAS ou fichiers texte) ou comme des instructions d'ordinateur (.COM). Les fichiers binaires peuvent être des suites de bits pour un affichage graphique, des données collectées à partir d'un périphérique, convertisseur analogique-numérique (compteur, thermomètre, baromètre, anémomètre, etc.), des fichiers codés (EBCDIC, TTS), ou des fichiers cryptés intentionnellement pour empêcher un accès non autorisé. Dans tous les cas, le Turbo stocke, copie, ouvre, lit et écrit ces fichiers, mais c'est le programmeur qui garde la trace de leur nom et du type d'information qu'ils contiennent. Comme toutes les autres structures de données du Turbo, les fichiers doivent être caractérisés par un identificateur et un type de données.

Voici un concept qui peut aider à la maîtrise de la gestion d'un fichier Turbo : alors que les fichiers peuvent devenir des collections de données vastes et complexes, les données doivent être manipulées élément par élément. Les fichiers de caractères sont lus et écrits caractère par caractère, les fichiers d'enregistrements, enregistrement par enregistrement. De même, les fichiers de données purement binaires sont manipulés octet par octet et les fichiers particuliers, bloc par bloc. La seule exception est le fichier texte, qui peut être lu caractère par caractère et également ligne par ligne.

Lignes

Une ligne n'est pas une structure de données comme une chaîne ou un tableau ; c'est une simple convention de gestion et non pas un élément de donnée. Elle n'est pas déclarée et n'emploie pas d'identificateurs mais fournit simplement une gestion du "flot " des mots dans la manipulation de fichiers texte. En définition Turbo, une ligne est un groupe de caractères terminé par deux caractères spéciaux : la combinaison retour chariot/passage à la ligne (Ctrl-M, Ctrl-J). Les fichiers texte sont lus soit caractère par caractère, soit ligne par ligne.

Opérations sur les fichiers

Les fichiers sont des structures de données complexes ; ils doivent être identifiés et tapés avec les éléments de données et les structures qui les composent. Un fichier est lu ou écrit uniquement par l'intermédiaire de ses éléments et ce processus est effectué élément par élément.

Les fichiers peuvent être créés ou effacés. On peut y écrire ou y lire des données puis les déplacer dans un autre fichier. De plus, les fichiers sont ouverts en début d'opération et fermés à la fin. Certains fichiers peuvent être lus et écrits en même temps ; d'autres doivent être fermés et réouverts entre ces deux opérations.

Le Turbo n'offre pas les procédures les plus courantes du Pascal Standard, GET et PUT ; cette omission est une bénédiction, car le Turbo emploie sur les fichiers les procédures WRITE et READ qu'il utilise pour les autres entrées/sorties. La Figure 4.25 montre la représentation classique d'un fichier.

```
Elément 1    Elément 2    Elément 3    Elément 4    Elément 5    EOF
```

Figure 4.25 : Représentation d'un fichier.

Tous les types de fichiers sont manipulés à l'aide du même jeu de procédures. Il est impossible d'entrer ou de sortir directement des données booléennes ou créées par l'utilisateur ; de même, on ne peut pas accéder directement à un nom de fichier, il faut l'assigner à une variable de fichier à l'aide de la procédure ASSIGN. La Figure 4.26 indique les bonnes et mauvaises manières d'assigner un nom à une variable de fichier.

```
(A)
   ASSIGN (VariableFichier, 'INFOCLUB.LST');
   RESET (VariableFichier);

(B)
   WRITELN ('Nom du fichier à lire :');
   READLN (NomFich);
   ASSIGN (VariableFichier, NomFich);
   RESET (VariableFichier);

(C)
   RESET ('INFOCLUB.LST');
```

Figure 4.26 : Assignation d'un nom de fichier à une variable de fichier.

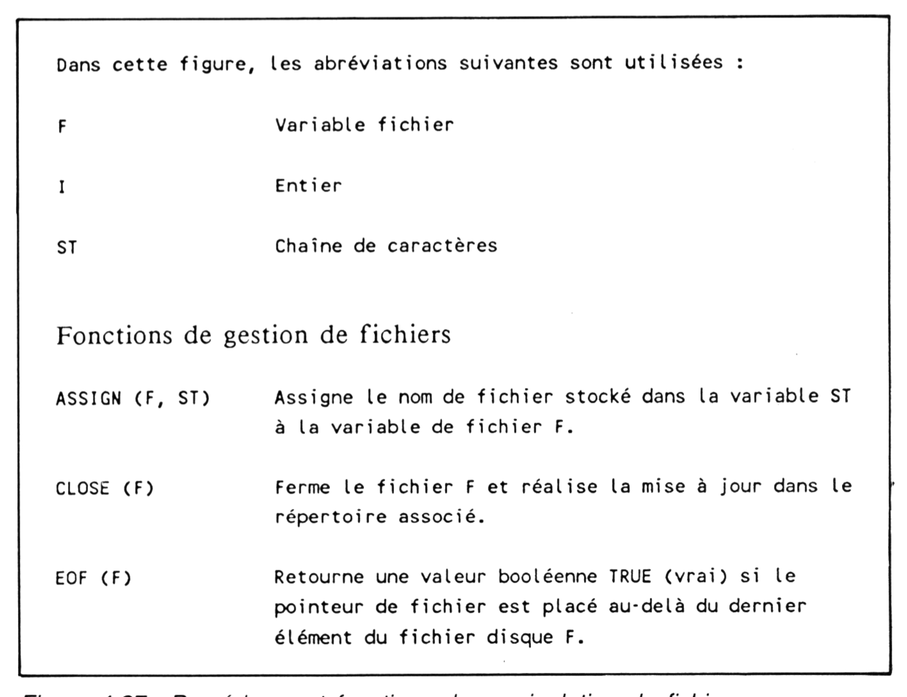
L'exemple A ne nécessite pas d'interaction de l'opérateur, alors que l'exemple B sollicite une entrée ; les deux sont corrects. L'exemple C génère des messages d'erreur car il est impossible de réinitialiser une chaîne ; on peut employer la procédure RESET uniquement sur une variable de fichier. RESET ouvre un fichier en lecture seule et se place au début du fichier. Pour l'écriture, on utilise la procédure REWRITE.

REWRITE crée un fichier s'il n'existe pas encore ; dans le cas contraire, le fichier existant est effacé.

Lorsqu'il a été ouvert avec une procédure RESET ou REWRITE, un fichier doit être fermé avec une procédure CLOSE. Si l'on oublie de fermer un fichier après sa lecture, le Turbo n'affiche pas de message d'erreur et cela ne pose généralement pas de problème. Par contre, si l'on oublie de le fermer après son écriture, on s'aperçoit après sortie du programme que le fichier est vide (0 octets dans le répertoire CP/M). La commande CLOSE donne au Turbo les instructions requises par le système d'exploitation pour "replacer" le fichier sur le disque et indiquer la nouvelle taille et la date de mise à jour.

La Figure 4.27 est un résumé de toutes les commandes de manipulation de fichiers. Il ne faut pas oublier que le Turbo ne se conforme pas à l'usage de GET et PUT du Pascal standard.

Le premier type de fichier abordé est le fichier à accès séquentiel, dans lequel on accède dans l'ordre aux éléments de données.



Dans cette figure, les abréviations suivantes sont utilisées :

F	Variable fichier
I	Entier
ST	Chaîne de caractères

Fonctions de gestion de fichiers

ASSIGN (F, ST)	Assigne le nom de fichier stocké dans la variable ST à la variable de fichier F.
CLOSE (F)	Ferme le fichier F et réalise la mise à jour dans le répertoire associé.
EOF (F)	Retourne une valeur booléenne TRUE (vrai) si le pointeur de fichier est placé au-delà du dernier élément du fichier disque F.

Figure 4.27 : Procédures et fonctions de manipulation de fichiers.

EOLN (F)	Retourne une valeur booléenne TRUE (vrai) si le pointeur de fichier est positionné sur le caractère "retour chariot" dans le fichier texte F. Dans un fichier texte, si EOF (F) est vrai, alors EOLN (F) est aussi vrai.
ERASE (F)	Supprime le fichier disque F.
FILEPOS (F)	Retourne la position en cours du pointeur de fichier de F ; le résultat est toujours un entier. Cette instruction ne peut pas être associée à des fichiers texte.
FILESIZE (F)	Retourne la taille du fichier disque F exprimée en nombre d'éléments de ce fichier (pour un fichier d'enregistrements, on obtient un nombre d'enregistrements) ; le résultat est toujours un entier. Cette instruction ne peut pas être associées à des fichier texte.
FLUSH (F)	Vide la mémoire tampon interne du fichier disque F ; cette opération est faite automatiquement et cette fonction est rarement utilisée par un programme. Cette instruction ne peut pas être associée à des fichiers texte.
RENAME (F, ST)	Renomme le fichier disque F avec le nom de fichier stocké dans la variable ST.
REWRITE (F)	Prépare le nouveau fichier disque F pour un traitement ; s'il existe déjà, le fichier F sera effacé puis créé à nouveau. Le pointeur de fichier est positionné au début.
RESET (F)	Prépare le fichier disque F (déjà créé) pour un traitement ; si le fichier n'existe pas, une erreur d'entrée/sortie est générée. Le pointeur de fichier est positionné au début.

Figure 4.27 (suite)

SEEK (F, I)	Déplace le pointeur de fichier sur le Ième composant du fichier F. Cette instruction ne peut pas être associée à des fichiers texte.
SEEKEOF (F)	Retourne la valeur booléenne TRUE (vrai) si le pointeur de fichier est positionné au-delà du dernier composant du fichier F. A l'inverse de EOF, SEEKEOF écarte les blancs, les tabulations et les marques de fin de ligne avant de tester la fin du fichier.
SEEKEOLN (F)	Retourne la valeur booléenne TRUE (vrai) si le pointeur de fichier est positionné dans le fichier texte F sur un caractère "retour chariot". A l'inverse de EOLN, SEEKEOLN écarte les blancs et les tabulations avant de tester le caractère "retour chariot". Dans un fichier texte, si EOF (F) est vrai, alors SEEKEOLN (F) est aussi vrai.

Figure 4.27 (suite)

La Figure 4.28 montre un programme de manipulation de fichiers séquentiels écrit pour aider à l'élaboration de cet ouvrage. Le traitement de texte XyWrite a été utilisé (XyWrite est au traitement de texte ce que le Turbo est aux langages, c'est-à-dire le meilleur et le plus rapide), mais l'éditeur avait besoin d'un fichier ASCII sans commandes XyWrite imbriquées pour son système de photocomposition informatisé. Cependant, le fichier initial comportait de nombreux codes de contrôle XyWrite pour générer des recopies d'écran et des affichages sophistiqués (double interligne, large marge, alignement de tables, etc.). Le programme de la Figure

4.28 lit le fichier XyWrite caractère par caractère, avec un fichier texte non compilé sans enregistrements ni tableaux, avec seulement des milliers de caractères.

Il teste le début ([) et la fin (]) de chaque commande XyWrite. Les caractères lus après le caractère de début de commande ne sont pas écrits dans le fichier texte de sortie tant que le programme n'a pas dépassé le caractère de fin de commande. Sans commandes imbriquées, le programme serait beaucoup plus simple. Lors de la présence du symbole de début de commande, une variable booléenne (signal) serait initialisée. Le programme pourrait accéder à chaque caractère et, tant que l'état du signal ne serait pas modifié, le caractère ne serait pas écrit. A la rencontre du symbole de fin de commande, le signal serait modifié et le programme reprendrait la sortie de caractères. Avec les commandes imbriquées, deux ou trois délimiteurs de fin de commande peuvent être rencontrés sans qu'aucun d'eux soit directement associé à celui de la commande d'ouverture. Voici le détail de l'opération :

... texte à imprimer [début de la commande [commande imbriquée *] suite et fin de la commande !] suite du texte à imprimer.

Un simple test booléen s'initialiserait de lui-même lorsqu'il rencontrerait le premier caractère de fin de commande (il apparaît ci-dessus précédé d'un astérisque) et afficherait une partie de la chaîne de commandes. Nous voulons que le programme continue à supprimer les sorties jusqu'à ce qu'il rencontre le symbole de fin de commande associé au premier caractère de début de commande (précédé par le point d'exclamation). Nous devons garder une trace du nombre de caractères de début de commande rencontrés et ne pas reprendre la sortie tant que le dernier caractère de fin de commande n'est pas rencontré.

Le programme de la Figure 4.28 emploie un compteur pour garder une trace des couples début de commande-fin de commande. Cela est caractéristique de nombreuses applications de traitement de texte.

Dans la section de déclaration, on voit que EntréeFichier et SortieFichier sont des variables de fichiers du type prédéfini TEXT. Les déclarations suivantes ont un effet identique :

EntreeFichier, SortieFichier : FILE OF CHAR ;
EntreeFichier, SortieFichier : TEXT ;

```
PROGRAM FiltreXyWrite;

VAR
   UnCar : CHAR;
   EntreeFichier, SortieFichier : TEXT; {type de fichier prédéfini}
   CarSpecial : INTEGER;
   Entrezie, Sortzie : STRING[4];

PROCEDURE Initialise;
BEGIN
   WRITE ('Nom du fichier contenant les codes XyWrite :');
   READLN (Entrezie);
   WRITE ('Nom du fichier de sortie :');
   READLN (Sortzie);
   ASSIGN (EntreeFichier, Entrezie);
   ASSIGN (SortieFichier, Sortzie);
   RESET (EntreeFichier);
   REWRITE (SortieFichier);
END;

PROCEDURE Epuration;
BEGIN
   WHILE NOT EOF (EntreeFichier) DO
      BEGIN
         READ (EntreeFichier, UnCar);
         IF UnCar = '[' THEN CarSpecial := CarSpecial + 1;
         IF UnCar = ']' THEN
            BEGIN
               CarSpecial := CarSpecial - 1;
               READ (EntreeFichier, UnCar);
               IF UnCar = '[' THEN CarSpecial := CarSpecial + 1;
            END;
         IF CarSpecial = 0 THEN
            BEGIN
               WRITE (SortieFichier, UnCar);
               WRITE (UnCar) {écho sur l'écran pour la mise au point}
            END;
      END;
END;
```

Figure 4.28 : Programme illustrant l'utilisation d'un fichier texte pour l'entrée et la sortie.

```
BEGIN
   Initialise;
   Epuration;
   CLOSE (EntreeFichier);
   CLOSE (SortieFichier);
   WRITE ('Traitement terminé !');
END.
```

Figure 4.28 (suite)

Les variables Inzie et Outzie servent à stocker les noms des fichiers manipulés par le programme. Il est impossible d'inclure l'identificateur de disque dans la chaîne (par exemple B:LETTRE.TXT). Le fichier d'entrée, qui est uniquement lu, commence par RESET alors que le fichier de sortie est à la fois créé et commencé par la commande REWRITE. Les deux fichiers sont fermés à la fin du programme. Le résultat est écrit sur le fichier disque tel qu'il aurait été envoyé sur une imprimante, sur un écran ou sur n'importe quel autre périphérique :

WRITE (SortieFichier, UnCar) ;
WRITE (UnCar) ; {**Envoi du caractere traite sur l'ecran**}

Dans l'élaboration de programmes pour la lecture et l'écriture de fichiers, la technique d'affichage des entrées et des sorties sur l'écran est utile pour la mise au point. Les lignes qui permettent l'affichage sur l'écran peuvent être supprimées lorsque le programme tourne correctement. La Figure 4.29 montre un fichier XyWrite classique utilisant des commandes délimitées par des crochets carrés ([et]). La Figure 4.30 montre le même fichier après filtrage des caractères de contrôle par le programme de la Figure 4.28.

Le programme utilise les commandes READ et WRITE plutôt que les commandes READLN et WRITELN de manière à lire le fichier caractère par caractère. READLN est plus rapide, mais uniquement si les lignes du fichier texte cadrent parfaitement avec la taille de la variable de chaîne dans laquelle on les lit. La

```
[RHA

[FC]Mise en oeuvre du Turbo Pascal
Introduction

][RFA



[FC]Page [PN]
[FL]
] [FD66] [PL60] [LM17] [RM70]
Introduction :
[IP5,0] [LS2]
[MDBO]Utilité de cet ouvrage :
[MDNM]
Ce livre est destiné à aider le lecteur en lui donnant aussi rapidement
que possible les moyens de résoudre un problème quelconque grâce au
[MDUL] TURBO[MDNM] Pascal. Il découvrira la [MDUL] programmation
structurée[MDNM] et ses nombreux avantages ; la Figure 1.2 montre un
programme type.
[LM12] [RM75] [IP0,0]
[NB]/p/
[MDBO] (Figure 1.2)[MDNM]
```

Figure 4.29 : Fichier généré par le traitement de texte XyWrite.

```
Introduction :

Utilité de cet ouvrage :

Ce livre est destiné à aider le lecteur en lui donnant aussi rapidement
que possible les moyens de résoudre un problème quelconque grâce au
TURBO Pascal. Il découvrira la programmation
structurée et ses nombreux avantages ; la Figure 1.2 montre un
programme type.

/p/
(Figure 1.2)
```

Figure 4.30 : Fichier de la Figure 4.29 traité par le programme de la Figure 4.28.

Figure 4.31 propose un programme qui illustre les différentes manières de lire un fichier texte. On remplace le fichier LETTRE.TXT par n'importe quel petit fichier texte généré par un autre traitement de texte ou par un programme tel que l'éditeur de texte CP/M ED. Ensuite, on modifie la longueur de la chaîne appelée UneLigne (255 caractères, maximum autorisé) en lui attribuant des valeurs plus petites (5, 15, 40, 80), et on exécute le programme.

```
PROGRAM LectureSimpleFichierTexte;

VAR
   UneLigne : STRING[255];
   UnCar : CHAR;
   EntreeFichier, SortieFichier : TEXT;
   PasCar : INTEGER;

PROCEDURE Initialise;
BEGIN
   ASSIGN (EntreeFichier, 'LETTRE.TXT');
   ASSIGN (SortieFichier, 'FIN.TXT');
   RESET (EntreeFichier);
   REWRITE (SortieFichier);
   PasCar := 0
END;

PROCEDURE LitLigne;
BEGIN
   WHILE NOT EOF (EntreeFichier) DO
      BEGIN
         READ (EntreeFichier, UneLigne);
         WRITE (UneLigne);
         WRITE (SortieFichier, UneLigne);
         READLN (EntreeFichier, UneLigne);
         WRITELN (UneLigne);
         WRITELN (SortieFichier, UneLigne);
      END;
END;
```

Figure 4.31 : Programme illustrant la lecture de fichiers texte par caractère et par ligne.

```
PROCEDURE LitCar;
BEGIN
   WHILE NOT EOF (EntreeFichier) DO
      BEGIN
         READ (EntreeFichier, UnCar);
         WRITE (UnCar);
         WRITE (SortieFichier, UnCar);
      END;
END;

PROCEDURE LitCarPlusLigne;
BEGIN
   WHILE NOT EOF (EntreeFichier) DO
      BEGIN
         READ (EntreeFichier, UnCar);
         WRITE (SortieFichier, UnCar);
         WRITELN (UnCar);

         IF EOLN (EntreeFichier) THEN
            WRITELN ('*** Fin de ligne rencontrée ***');
      END;
END;

BEGIN
   Initialise;
   LitLigne;
   CLOSE (SortieFichier);
   RESET (EntreeFichier);
   REWRITE (SortieFichier);
   LitCar;
   CLOSE (SortieFichier);
   RESET (EntreeFichier);
   REWRITE (SortieFichier);
   LitCarPlusLigne;
   CLOSE (EntreeFichier);
   CLOSE (SortieFichier);
   WRITE ('Traitement terminé !');
END.
```

Figure 4.31 (suite)

Le programme emploie les identificateurs standard booléens EOF *(End Of File*, fin de fichier) et EOLN *(End Of LiNe*, fin de ligne). Etant donné que les fichiers n'ont pas une taille prédéterminée, ces identificateurs offrent la meilleure méthode pour savoir si l'on a atteint la fin d'un fichier. Ils se prêtent à de nombreuses structures de contrôle de programme ; ils prennent la valeur booléenne TRUE (vrai) lorsque l'indicateur de fin de fichier ou de fin de ligne est rencontré. La fonction EOF fournit la méthode la plus courante pour contrôler l'entrée à partir d'un fichier dont la taille est inconnue ou sans importance. Cependant, certaines situations impliquent la connaissance exacte du nombre d'enregistrements composant un fichier, par exemple lors d'un programme de tri. Les techniques de détermination de la taille de fichiers seront brièvement traitées.

Il est conseillé à l'utilisateur d'exécuter ce programme avec ses propres fichiers. On remarque que le Turbo n'autorise pas la lecture et l'écriture simultanées d'un fichier texte. Celui-ci doit être fermé et réouvert entre ces deux opérations ; cette restriction ne s'applique pas aux autres fichiers.

Fichiers non-texte

Cet ouvrage a commencé par traiter les fichiers texte (fichiers particuliers) avant les fichiers d'un type plus général, parce que la capacité à déplacer des caractères vers ou à partir d'un fichier puis de visualiser la modification (en utilisant la commande DOS TYPE pour afficher le contenu d'un fichier ASCII) aide à la compréhension du processus d'ouverture et de fermeture de fichier.

Les fichiers non-texte sont des fichiers quelconques n'appartenant pas au type prédéfini TEXT. La Figure 4.32 montre uniquement les lignes de code du programme de la Figure 4.22 qui traitent de l'écriture et de la lecture de fichiers de données. Les commentaires de cette figure expliquent le déroulement du processus.

Recherche dans des fichiers

Une des manières les plus rapides de rechercher un élément spécifique dans un groupe ordonné (trié) consiste à effectuer une recherche dichotomique simple. On compare l'élément cible avec un élément choisi au milieu du fichier. Si la cible est plus importante que cet élément, on sait que la recherche doit se situer dans la moitié supérieure du fichier et non dans la moitié inférieure. On

```
PROGRAM Demo_Enregistrement;

(Section de déclaration normale, avec définition de
l'enregistrement EnregMembre)
.
.
.
VAR
   EntreeFichier, SortieFichier : FILE OF EnregMembre;

(Les fichiers sont constitués d'enregistrements définis
plus haut)
.
.
.
   WRITELN ('Nom du fichier contenant les noms :');
   READLN (Reponse);
   ASSIGN (SortieFichier, Reponse);
   REWRITE (SortieFichier);

(Assignation à un fichier et ouverture pour écriture)

   WHILE NOT ToutFait DO
   BEGIN
      WRITELN ('Nom :');
      READLN (Nom);
      Membre.Cle := Nom;
      WRITELN ('Appel :');
      READLN (Appel);
      Membre.AppelSigne := Appel;

(Cette zone correspond à l'entrée de données pour les éléments
de l'enregistrement EnregMembre)

      Membre.Niveau := Ticket;
      WRITE (SortieFichier, Membre);

(L'enregistrement est écrit dans le fichier lorsque tous les éléments ont
été entrés. Un fichier d'enregistrements est rempli avec un enregistrement
à la fois)

   CLOSE (SortieFichier);    (ne pas oublier)
END;

PROCEDURE Lit_Fichier_Existant;
BEGIN
   WRITELN ('Nom du fichier contenant les enregistrements du club :');
```

Figure 4.32 : Fragment de programme illustrant l'écriture d'enregistrements dans des fichiers de données.

```
    READLN (Reponse);
    ASSIGN (EntreeFichier, Reponse);
    RESET (EntreeFichier);
    WHILE NOT EOF (EntreeFichier) DO
    BEGIN
       READ (EntreeFichier, Membre);
       WITH Membre DO
       BEGIN

{Le fichier est ouvert en lecture par la commande RESET. La taille du
fichier n'étant pas connue, la fonction EOF permet un déplacement
d'enregistrement en enregistrement jusqu'à la fin du fichier. Les
données sont lues comme elles sont écrites, un enregistrement à la fois}

.
.
.
    END;
    CLOSE (EntreeFichier);
END;
.
.
.
```

Figure 4.32 (suite)

compare ensuite la cible avec un élément de fichier choisi au milieu de la moitié supérieure, ce qui permet d'affiner la recherche. Même dans une liste de 16 000 éléments, on accède généralement à l'élément recherché en moins de 16 opérations. Pour appliquer cette technique à un fichier, il faut connaître précisément le nombre d'enregistrements qu'il contient.

La Figure 4.33 illustre l'application d'une recherche dichotomique à un tableau très important (entiers compris entre 0 et 16 000) stocké en mémoire. La Figure 4.35 applique cette technique à un vaste fichier de données externes en employant une fonction qui détermine la taille précise du fichier avant le début de la recherche.

```
PROGRAM RechercheDichotomique;

VAR
   Minimum, Maximum, Inconnu, Passe, Estime : INTEGER;
   Trouve : BOOLEAN;

PROCEDURE Initialise;
BEGIN
   CLRSCR;
   Trouve := FALSE;
   Passe := 1;
   Estime := 0;
   Minimum := 0;
   Maximum := 16000;
   Inconnu := RANDOM (Maximum);
   WRITELN ('Je pense à un nombre compris entre 1 et ',Maximum);
   WRITELN ('pouvez-vous le deviner ?');
END;

PROCEDURE AfficheEtat;
BEGIN
   WRITE ('Essais n ', Passe:2, ' vous avez estimé que le nombre est ');
   WRITELN ('compris entre ', Minimum:5, ' et ', Maximum:5, '.');
END;

PROCEDURE Evaluation;
BEGIN
   Estime := (Minimum + Maximum) DIV 2; {détermination du milieu}
   WRITELN;
   WRITELN ('Votre estimation initiale est : ', Estime);
   WRITELN;
   WHILE NOT Trouve DO
   BEGIN
      IF Estime = Inconnu THEN
         BEGIN
            WRITELN ('Le nombre à deviner est : ', Inconnu);
            Trouve := TRUE;
         END;
      IF Estime < Inconnu THEN {il se trouve dans la moitié supérieure}
         BEGIN
            Passe := Passe + 1;
            Minimum := Estime; {la limite inférieure prend la valeur estimée}
            Estime := (Minimum + Maximum) DIV 2;
            AfficheEtat;
         END;
      IF Estime > Inconnu THEN {il se trouve dans la moitié inférieure}
         BEGIN
```

Figure 4.33 : Programme de démonstration d'une recherche dichotomique rapide dans un tableau.

```
                Passe := Passe + 1;
                Maximum := Estime; {la limite supérieure prend la valeur estimée}
                Estime := (Minimum + Maximum) DIV 2;
                AfficheEtat;
            END;
        END;
    END;

    BEGIN
        Initialise;
        Evaluation;
        WRITELN;
        WRITELN ('Il a fallu', Passe:3, ' essais pour deviner le nombre.');
        WRITELN;
    END.
```

Figure 4.33 (suite)

Ce programme est si rapide qu'on a peine à imaginer qu'il effectue réellement la recherche. Pour en avoir la preuve, on modifie MaxElément de manière à observer la différence de durée entre une recherche sur 100 valeurs et une recherche sur 16 000 valeurs ; la Figure 4.34 montre le résultat de deux déroulements.

En résumé, la recherche dichotomique est très rapide lorsqu'elle concerne les données stockées en mémoire. Elle est également performante lorsqu'il s'agit de recherches d'enregistrements individuels dans des fichiers de données externes importants.

Une des approches de la recherche dans des fichiers consiste à lire successivement chaque enregistrement et à extraire un champ clé (nom, numéro de client, etc.) de chacun d'entre eux pour le stocker en mémoire sous la forme d'un tableau à deux dimensions avec le numéro d'enregistrement. Le tableau est ensuite scruté, puis on accède à l'enregistrement dont le numéro correspond. Cette méthode est assez efficace, mais seulement si le tableau tient en mémoire. Étant donné que le Turbo limite à 64 K la quantité de données qu'il peut stocker, il faut trouver un type de recherche adapté à des fichiers importants stockés sur disque.

L'efficacité de la recherche dichotomique se prête aux opérations de recherche dans des fichiers séquentiels qui peuvent contenir des milliers d'enregistrements. Pour les recherches sur fichiers disque, il faut disposer de procédures qui déterminent le numéro d'enregistrement le plus élevé dans le fichier sans avoir à lire et

```
Je pense à un nombre compris entre 1 et 16000
pouvez-vous le deviner ?

Votre estimation initiale est : 8000

Essais n  2 vous avez estimé que le nombre est compris entre     0 et  8000.
Essais n  3 vous avez estimé que le nombre est compris entre  4000 et  8000.
Essais n  4 vous avez estimé que le nombre est compris entre  6000 et  8000.
Essais n  5 vous avez estimé que le nombre est compris entre  6000 et  7000.
Essais n  6 vous avez estimé que le nombre est compris entre  6500 et  7000.
Essais n  7 vous avez estimé que le nombre est compris entre  6750 et  7000.
Essais n  8 vous avez estimé que le nombre est compris entre  6875 et  7000.
Essais n  9 vous avez estimé que le nombre est compris entre  6875 et  6937.
Essais n 10 vous avez estimé que le nombre est compris entre  6906 et  6937.
Essais n 11 vous avez estimé que le nombre est compris entre  6921 et  6937.
Essais n 12 vous avez estimé que le nombre est compris entre  6921 et  6929.
Essais n 13 vous avez estimé que le nombre est compris entre  6921 et  6925.
Essais n 14 vous avez estimé que le nombre est compris entre  6923 et  6925.
Le nombre à deviner est : 6924

Il a fallu 14 essais pour deviner le nombre.

Je pense à un nombre compris entre 1 et 16000
pouvez-vous le deviner ?

Votre estimation initiale est : 8000

Essais n  2 vous avez estimé que le nombre est compris entre  8000 et 16000.
Essais n  3 vous avez estimé que le nombre est compris entre 12000 et 16000.
Essais n  4 vous avez estimé que le nombre est compris entre 14000 et 16000.
Essais n  5 vous avez estimé que le nombre est compris entre 15000 et 16000.
Essais n  6 vous avez estimé que le nombre est compris entre 15000 et 15500.
Essais n  7 vous avez estimé que le nombre est compris entre 15250 et 15500.
Essais n  8 vous avez estimé que le nombre est compris entre 15375 et 15500.
Essais n  9 vous avez estimé que le nombre est compris entre 15437 et 15500.
Essais n 10 vous avez estimé que le nombre est compris entre 15468 et 15500.
Essais n 11 vous avez estimé que le nombre est compris entre 15468 et 15484.
Essais n 12 vous avez estimé que le nombre est compris entre 15468 et 15476.
Essais n 13 vous avez estimé que le nombre est compris entre 15472 et 15476.
Le nombre à deviner est : 15474

Il a fallu 13 essais pour deviner le nombre.
```

Figure 4.34 : Écran d'affichage généré par le programme de la Figure 4.33.

```
PROGRAM RechercheDichotomiqueAvecFichier;

TYPE
   EnregClient = RECORD
      Nom : STRING[13];
      Prenom : STRING [15];
      Rue : STRING[30];
      Ville : STRING[19];
      CodePost : STRING[5];
   END;

VAR
   EntreeFich : FILE OF EnregClient;
   Client : EnregClient;
   FichSource : STRING[14];
   SurNom : STRING[14];
   EnregNum, MaxEnreg : INTEGER;
   Toutfini : BOOLEAN;
   Encore : CHAR;
   Minimum, Maximum, Cherch, Dercherch : INTEGER;
   Trouve, FaitAutre : BOOLEAN;

PROCEDURE ChercheUnNom;
BEGIN
   CLRSCR;
   WRITELN ('Nom du fichier à scruter :');
   READLN (FichSource);
   ASSIGN (EntreeFich, FichSource);
   RESET (EntreeFich);
   Trouve := FALSE;
   MaxEnreg := FILESIZE (EntreeFich);
   WRITELN ('Taille du fichier : ', MaxEnreg);
   FaitAutre := TRUE;
   WHILE FaitAutre DO
      BEGIN
         Minimum := 0;
         Maximum := MaxEnreg;
         WRITELN ('Nom recherché :');
         READLN (SurNom);
         WHILE NOT Trouve DO
            BEGIN
               Cherch := (Minimum + Maximum) DIV 2;
                              {coeur de la recherche dichotomique}
               IF Cherch = DerCherch THEN Minimum := MAXINT;
               Dercherch := Cherch;
               SEEK (EntreeFich, Cherch); {pointe l'enregistrement}
               READ (EntreeFich, Client); {lit l'enregistrement complet}
```

Figure 4.35 : Programme d'exécution d'une recherche dichotomique très rapide sur un fichier de données externes.

```
            IF SurNom = Client.Nom THEN
               BEGIN (affiche l'enregistrement complet s'il est trouvé)
                  Trouve := TRUE;
                  CLRSCR;
                  WRITELN (Client.Nom, '  ', Client.Prenom);
                  WRITELN (Client.Rue);
                  WRITELN (Client.Ville, '  ', Client.CodePost);
                  WRITELN;
                  WRITELN ('Voulez-vous chercher un autre nom ? O/N');
                  READLN (KBD, Encore);
                  IF (Encore = 'O') OR (Encore = 'o') THEN
                     BEGIN
                        FaitAutre := TRUE;
                        Trouve := FALSE;
                        CLRSCR;
                        WRITELN ('Nom recherché :');
                        READLN (SurNom);
                        Minimum := 0;
                        Maximum := MaxEnreg;
                     END
                  ELSE FaitAutre := FALSE;
               END;
            IF Minimum < Maximum THEN (tant que la recherche peut s'effectuer)
               BEGIN
                  IF SurNom > Client.Nom THEN Minimum := Cherch;
                  IF SurNom < Client. Nom THEN Maximum := Cherch;
               END
            ELSE (tous les enregistrements ont été scrutés sans succès)
               BEGIN
                  WRITELN ('Je ne trouve pas ce nom !');
                  WRITELN;
                  FaitAutre := TRUE;
                  Trouve := FALSE;
                  WRITELN ('Nom recherché :');
                  READLN (SurNom);
                  Minimum := 0;
                  Maximum := MaxEnreg;
               END;
         END;
         CLOSE (EntreeFich);
      END;
END;

BEGIN
   ChercheUnNom;
END.
```

Figure 4.35 (suite)

à trier successivement tous les enregistrements. De plus, on doit disposer d'une procédure qui accède uniquement à l'enregistrement contenant l'information cible.

Le Turbo offre des outils adaptés à ces opérations. La fonction FILESIZE donne instantanément le nombre d'enregistrements d'un fichier car elle n'a pas besoin de les lire, puisque le système d'exploitation place cette information dans le bloc de contrôle du fichier. De façon similaire, la fonction SEEK recherche, suivant le numéro indiqué, l'enregistrement que l'on veut lire.

Le programme de la Figure 4.35 est conçu pour effectuer la maintenance d'une liste d'adresses d'entreprises comprenant un millier de noms. Il faut attacher une importance particulière à la syntaxe des fonctions FILESIZE, SEEK et READ. FILESIZE et SEEK ont toutes deux besoin d'un nom de fichier comme argument ; SEEK requiert également un numéro d'enregistrement. Le temps nécessaire à la recherche d'un numéro d'enregistrement particulier est négligeable. Un écran pour le résultat du déroulement de ce programme n'a pas d'intérêt, car il ne rendrait pas compte de la vitesse d'exécution. L'utilisateur s'apercevra, en employant cette technique avec ses propres fichiers de données, qu'elle est très rapide.

Fichiers sans type

Le Turbo offre un autre type de fichiers appelé *sans type* . Il permet de déplacer des blocs de 128 octets dans une variable d'un fichier sans type (à l'aide des procédures BLOCKREAD et BLOCKWRITE). Ce type d'opération est contraire à l'esprit du Pascal, car il s'agit d'un transfert mécanique des octets sans aucune considération pour les types de données qu'ils contiennent. C'est une application "exotique " pour ceux qui écrivent leurs propres gestionnaires d'entrées/sorties, schémas de protection de copies et de programmes optimisés pour travailler en liaison étroite avec le système d'exploitation. Ce sujet ne fait pas l'objet de cet ouvrage.

Fichiers : conclusion

La manipulation de fichiers est fondamentale pour des programmes sophistiqués relatifs à la plupart des opérations scientifiques, de gestion et de traitement de texte. La seule manière de maîtriser le jeu de procédures du Turbo consiste à faire des

essais avec des petits fichiers et à étudier et modifier les exemples de programmes de ce chapitre. Tous les éléments fondamentaux de la création, de l'écriture et de la lecture des fichiers Turbo ont été abordés.

ENSEMBLES

Le Turbo Pascal utilise le concept d'*ensemble* ; ce concept est facilement compréhensible, mais limité en ce qui concerne les applications pratiques. On sait déjà que tous les identificateurs doivent avoir un type de données spécifié mais qu'ils peuvent être déterminés optionnellement comme appartenant également à un ensemble défini par l'utilisateur. Ensuite, les variables peuvent être testées comme faisant partie de cet ensemble.

En théorie, les ensembles forment un outil puissant. En pratique, ils sont employés peu fréquemment alors qu'ils peuvent être très utiles selon l'ingéniosité des utilisateurs. Parmi la quantité de possibilités offertes par le Turbo, les ensembles sont souvent oubliés et les programmeurs n'ont pas l'habitude de maîtriser ce concept.

Le Turbo restreint le nombre d'éléments d'un ensemble à 256. Cependant, la limite la plus importante tient au fait que les ensembles peuvent uniquement être composés de types de données scalaires non structurés (et comme toujours, pas de nombres réels) qui ont un prédécesseur et un successeur immédiats ; cela implique de sévères restrictions. Par exemple, il est impossible d'élaborer des ensembles de chaînes ou des ensembles composés d'éléments tels que des codes postaux (la plupart de ces codes excèdent MaxInt, 32567) et de nombreuses autres valeurs numériques. En fait, les ensembles doivent être associés à un nombre d'éléments discrets, généralement pour tester ou filtrer des entrées/sorties de caractères.

Un ensemble est utile quand une gamme limitée d'entrées peut être définie comme telle. Les entrées sont testées dans l'ensemble par une instruction et non par une fastidieuse série de comparaisons élément par élément. Une autre application fréquente consiste, dans les programmes de conversion de codes, à comparer les entrées avec plusieurs ensembles : l'ensemble de caractères texte, celui des caractères de contrôle et celui des caractères à éliminer.

Après le test d'appartenance à un ensemble, on peut choisir différentes procédures de traitement.

Cependant, l'utilisation d'ensembles n'offre pas plus de possibilités que les tests conventionnels et les structures de contrôle. Il suffit de choisir les applications pour lesquelles les ensembles sont les plus appropriés. L'application la plus courante consiste à filtrer les entrées non autorisées du clavier ; la Figure 4.36 montre un menu correspondant à ce type d'application. Les entrées non valides sont prises en charge par la clause ELSE de l'instruction CASE. Tout caractère qui ne se situe pas dans la gamme 1 à 8 entraîne l'affichage d'un message généré par la clause ELSE

```
PROCEDURE Choix;              {Permet de définir un choix dans un menu}
BEGIN
   READ (KBD, Choix);
   Option := ORD (Choix) - 48; {convertit un caractère ASCII en entier}
   CASE Option OF              {options accessibles avec un chiffre}
      1: Accepte_Nouv_Noms;
      2: Cherche_Un_Nom;
      3: Liste_Alphabetique;
      4: Affiche_Liste_Sur_Ecran;
      5: Genere_Etiquettes;
      6: Cree_List_Adresses;
      7: Lit_Repertoire;
      8: Sortie;
   ELSE
      BEGIN
         GOTOXY (19, 22);
         WRITELN (#7, '*** choix impossible, recommencez ***');
         GOTOXY (25, 23);
         WRITELN ('Quel nombre choisissez-vous ?');.
         GOTOXY (55, 23);
         Choix;
      END; {fin de else}
   END; {fin de case}
END;
```

Figure 4.36 : Filtrage des entrées erronées à l'aide de la construction CASE/ELSE.

pour demander une nouvelle entrée. Cette technique est très efficace et très simple bien qu'elle n'emploie pas les ensembles.

Ce programme présente l'avantage supplémentaire d'exploiter le fichier KBD standard de telle sorte que l'opérateur n'ait pas besoin de taper la touche Return après avoir exécuté une sélection. Cette opération est similaire à celle de la fonction INKEY$ du BASIC et a été illustrée par la Figure 4.16.

Cette procédure est impossible si les choix du menu excèdent un nombre pouvant être représenté par un caractère unique. L'utilisation d'une touche alphabétique (A à Z) permet un choix parmi 26 possibilités (voir la Figure 4.16). Mais pour interpréter plus de 26 choix, il est préférable d'employer les ensembles. La Figure 4.37 montre le programme de la Figure 4.36 modifié pour utiliser la notion d'ensemble.

```
PROCEDURE Choix;
VAR
   Choix : SET OF BYTE;
   Option, Resultat : INTEGER;
   Encore : STRING[2];
BEGIN
   Choix := [1..8, 10, 15];
   READLN (Encore); {il faut taper Return pour valider l'entrée}
   VAL (Encore, Option, Resultat); {convertit la chaîne en entier}
   IF (Option IN Choix) THEN
      BEGIN
         CASE Option OF              {options accessibles avec un chiffre}
            1: Accepte_Nouv_Noms;
            2: Cherche_Un_Nom;
            3: Liste_Alphabetique;
            4: Affiche_Liste_Sur_Ecran;
            5: Genere_Etiquettes;
            6: Cree_List_Adresses;
            7: Lit_Repertoire;
            8: Sortie;
           10: WRITELN ('Choix 10');
           15: WRITELN ('Choix 15');
         END;
      END;
   ELSE
      BEGIN
         GOTOXY (19, 22);
         WRITELN (#7, '*** choix impossible, recommencez ***');
         GOTOXY (25, 23);
         WRITELN ('Quel nombre choisissez-vous ?');
         GOTOXY (55, 23);
         Choix;
      END; {fin de else}
END;
```

Figure 4.37 : Procédure de la Figure 4.36 modifiée pour employer des ensembles.

POINTEURS

Les pointeurs font partie des structures de données les plus complexes et les moins connues du Turbo. Leur fonctionnement n'est ni évident ni analogue à celui des structures naturelles des tableaux et des chaînes. Tant que l'on élabore des programmes destinés à manipuler des fichiers de données importants et complexes, les tableaux restent la structure la plus simple pour un tri et un accès rapide aux informations. Il serait stupide d'ajouter la complexité des pointeurs à n'importe quel programme qui pourrait se contenter de tableaux.

Cependant, les pointeurs constituent la seule structure permettant d'accéder à des éléments de données reliés les uns aux autres mais que l'on ne peut pas atteindre en ajoutant ou en soustrayant simplement d'un index une valeur connue (règle de l'utilisation des tableaux).

Dans la section précédente, nous avons dû étendre nos concepts de recherche dichotomique aux enregistrements de fichiers disque lorsque la taille du fichier devenait trop importante pour qu'il puisse être contenu dans un tableau (ou simplement parce que le chargement en mémoire nécessité par chaque recherche prenait trop de temps). Nous avons employé un périphérique comme un pointeur pour exécuter la recherche dichotomique. Dans cette recherche, les numéros d'enregistrements, dont le nombre maximal était déterminé par la fonction FILESIZE et pointé par la fonction SEEK, servaient de pointeurs pour les enregistrements choisis. Le Turbo offre un type de données qui a la même fonction sur n'importe quelle structure de données stockée en mémoire. Un *pointeur* est un type de données qui contient l'adresse d'une variable. Cependant, il ne contient pas la variable. Comme tous les types de données, les pointeurs doivent être déclarés par un identificateur et un type.

Une *liste* n'est pas une structure de données formelle ; par conséquent, les listes n'ont pas besoin d'être identifiées et déclarées. Néanmoins, il s'agit d'un concept important intimement lié aux pointeurs. Un pointeur n'a en général pas d'intérêt en lui-même. Il devient important dans la détermination de la position d'une variable en relation avec d'autres variables. Une *liste chaînée* est un groupe de variables dans lequel chaque variable contient un pointeur pour une autre variable. Les pointeurs sont les "chaînes " reliant un élément à un autre.

Les pointeurs sont presque toujours employés avec des enregistrements. Dans la plupart des applications, un enregistrement comporte quelques données et au moins un pointeur qui contient au minimum l'adresse d'un autre enregistrement. Les enregistrements comportant les données et l'adresse de leur prédécesseur ou de leur successeur immédiat sont appelés *liste à chaînage unique.* Plus complexes, les listes à *chaînage double* incluent l'adresse du successeur et du prédécesseur des données. Encore plus complexes, des structures appelées *arbres* peuvent être composées de combinaisons complexes de successeurs et de prédécesseurs avec les données.

La caractéristique la plus importante des pointeurs est leur aptitude à relier certaines données aux adresses d'autres éléments de données. Pour les listes alphabétiques simples, les pointeurs ne font que compliquer les choses. Cependant, ils sont inestimables pour la création de bases de données élaborées. Par exemple, pour réaliser une gestion de base de données, on pourrait stocker tous les enregistrements par numéro de client, chaque enregistrement ayant un pointeur dans un enregistrement associé, quelle que soit la clé de tri sélectionnée. Dans ce cas, bien que les enregistrements soient stockés en ordre ascendant par numéro de client, chaque enregistrement de contrat client a un pointeur dans le contrat suivant, chaque solde client a un pointeur dans le solde suivant, etc.

On pourrait également établir des pointeurs de telle sorte que chaque client de la région est ait un pointeur dans le client de la région est suivant. En traversant adroitement les listes, de pointeur en pointeur, des demandes complexes telles que "générer la liste de tous les clients de la région est ayant signé un contrat et possédant un solde supérieur à 400 000 F " peuvent être satisfaites de manière plus efficace qu'en effectuant des recherches consistant à lire chaque enregistrement de la base et à lui faire subir une série de tests. Au fur et à mesure que la taille et la complexité de la base s'accroissent, l'intérêt de l'utilisation des pointeurs augmente. Deux caractéristiques donnent aux pointeurs leur place unique dans la gamme des structures de données Turbo.

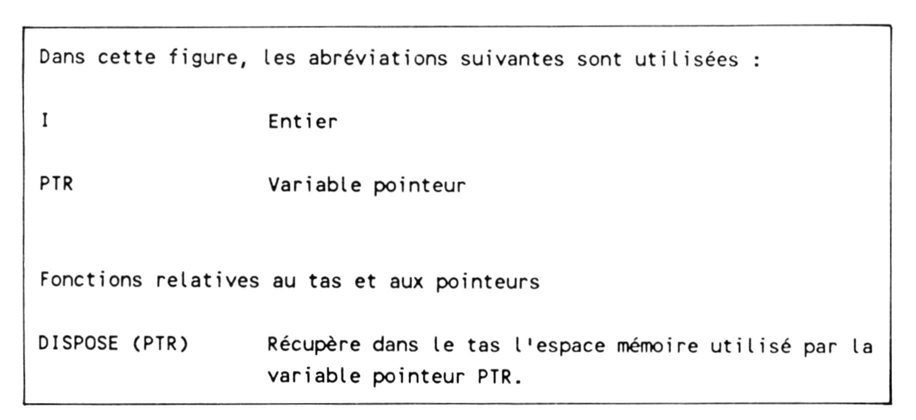
1. Les listes reliées par des pointeurs peuvent s'accroître de façon dynamique au cours de l'exécution d'un programme. Les listes sont les seules structures de données résidant en mémoire dont la taille puisse changer de manière dynamique. Par opposition,

bien que toutes les cellules d'un tableau ne soient pas forcément remplies, l'espace mémoire réservé à la taille maximale déclarée pour le tableau a été alloué au début de l'exécution du programme.

2. Un programme Turbo ne peut pas dépasser 64 K et la somme de toutes les structures de données (variables et tableaux) employées dans un programme quelconque sont soumises aux mêmes limites. Avec un ordinateur dont la mémoire totale est de 128 K, le plus gros programme Turbo ne peut pas dépasser cette taille.

Le tas est la quantité de mémoire restant après attribution de l'espace nécessaire au système d'exploitation, au programme et à toutes les variables. Cette zone mémoire sert au Turbo pour stocker les variables de pointeur créées dynamiquement. Dans les systèmes de moins de 128 K, il reste parfois un tas. Le tas est égal à l'espace laissé vacant par le programme et ses structures de données ; par exemple, si un programme n'occupe que 12 K et que ses données occupent seulement 4 K, il reste 112 K (128 – 16) réservés au tas.

La Figure 4.38 montre les procédures et les fonctions disponibles pour la manipulation des pointeurs et du tas. La plupart de ces fonctions dépassent l'objet de cet ouvrage mais sont incluses ici pour une simple référence.



Dans cette figure, les abréviations suivantes sont utilisées :

I	Entier
PTR	Variable pointeur

Fonctions relatives au tas et aux pointeurs

DISPOSE (PTR)	Récupère dans le tas l'espace mémoire utilisé par la variable pointeur PTR.

Figure 4.38 : Procédures et fonctions standard pour la manipulation des pointeurs et du tas.

FREEMEM (PTR, I)	Récupère dans le tas I octets de mémoire pour la variable pointeur PRT ; le nombre d'octets doit être exactement le même que celui qui a été préalablement affecté à la variable PTR par la fonction GETMEM.
GETMEM (PTR, I)	Alloue I octets de mémoire dans le tas pour la variable pointeur PTR.
MARK (PTR)	Assigne le pointeur de tas à la variable PTR.
MAXAVAIL	Retourne la taille du plus grand espace disponible dans le tas. Le résultat est toujours un entier mais permet plusieurs interprétations s'il est supérieur à MAXINT.
MEMAVAIL	Retourne le nombre de paragraphes (16 octets) libres dans le tas ; le résultat est toujours un entier.
NEW (PTR)	Crée le pointeur de variable PTR.
ORD (PTR)	Retourne l'adresse contenue dans le pointeur PTR ; le résultat est toujours un entier. Le résultat de cette fonction est différent si l'argument est un scalaire.
PTR (I)	Convertit la valeur entière I en un pointeur.
RELEASE (PTR)	Supprime toutes les variables dynamiques au-dessus de l'adresse PTR (PTR est préalablement initialisée par la procédure MARK).

Figure 4.38 (suite)

La Figure 4.39 montre un petit programme qui emploie des pointeurs. Il utilise les mêmes techniques de manipulation de chaînes que celles du programme de la Figure 4.11, en prenant cette fois-ci une chaîne composée de noms de détectives célèbres et en créant une liste de noms à chaînage unique. Dans ce type d'application, l'emploi de tableaux est une approche plus naturelle.

Cependant, la manipulation d'une petite quantité de données rend plus claires les utilisations de pointeurs et de listes.

Avant d'étudier le programme de la Figure 4.39, il faut observer la Figure 4.40, qui montre le résultat du déroulement de ce programme ; les relations entre les pointeurs sont évidentes.

```
PROGRAM DemoPointeur;

TYPE
   Nom = STRING[18];
   PointeurEnregDetective = ^EnregDetective;
   EnregDetective = RECORD
      Detective : Nom;
      DetectivePreced : PointeurEnregDetective;
   END;

CONST
  MaxEle = 5;

VAR
   Predecesseur, EnregDetectiveCourant, Temp : PointeurEnregDetective;
   I : INTEGER;
   UnNom : Nom;

PROCEDURE Gestion;
VAR
   PremCar, BarreOb, CarGauche : INTEGER;
   RangDonnees : STRING[73];

BEGIN
   RangDonnees := 'SHERLOCK HOLMES/MISS MARPLE/HERCULE POIROT/NERO WOLFE/INSPECTEUR MAIGRET/';
   PremCar := 1;
   BarreOb := 1;
   Predecesseur := NIL; {initialisation de la liste, le 1er pointeur à la fin}
   FOR I := 1 TO MaxEle DO
   BEGIN
      CarGauche := LENGTH (RangDonnees);
      BarreOb := POS ('/', RangDonnees);
      UnNom := COPY (RangDonnees, PremCar, (BarreOb - 1));
      RangDonnees := COPY (RangDonnees, (BarreOb + 1), (CarGauche - BarreOb));
      NEW (EnregDetectiveCourant);
      EnregDetectiveCourant^.Detective := UnNom;
      EnregDetectiveCourant^.DetectivePreced := Predecesseur;
      Predecesseur := EnregDetectiveCourant; {le prédécesseur pointe
                                 maintenant l'enregistrement courant}
      WRITELN ('Le détective en cours est : ', EnregDetectiveCourant^.Detective);
   END;
END;
```

Figure 4.39 : Programme de création d'une liste à chaînage unique à l'aide de pointeurs.

```
PROCEDURE Sortie;
BEGIN
   WRITELN;
   WRITELN ('Les noms lus à l aide de la série de pointeurs sont :');
   WRITELN;
   Temp := EnregDetectiveCourant;
   WHILE Temp <> NIL DO
   BEGIN
      WRITELN (Temp^.Detective);
      Temp := Temp^.DetectivePreced;
   END;
   WRITELN; WRITELN; WRITELN;
END;

BEGIN
   CLRSCR;
   Gestion;
   Sortie;
END.
```

Figure 4.39 (suite)

```
Le détective en cours est : SHERLOCK HOLMES
Le détective en cours est : MISS MARPLE
Le détective en cours est : HERCULE POIROT
Le détective en cours est : NERO WOLFE
Le détective en cours est : INSPECTEUR MAIGRET

Les noms lus à l aide de la série de pointeurs sont :

INSPECTEUR MAIGRET
NERO WOLFE
HERCULE POIROT
MISS MARPLE
SHERLOCK HOLMES
```

Figure 4.40 : Résultat du déroulement du programme de la Figure 4.39.

La procédure Gestion fait l'analyse syntaxique de la chaîne RangDonnées et donne les noms des détectives en créant une série d'enregistrements composés d'un nom de détective et d'un pointeur pour l'enregistrement qui caractérise le détective précédent. La procédure appelée Sortie examine le dernier enregistrement, affiche le nom du détective puis examine tour à tour les enregistrements précédents.

Bien que la même opération ait pu être réalisée avec un tableau simple (sans pointeurs) ou même avec un fichier texte simple (en le lisant jusqu'à la fin du fichier, EOF), il est important de remarquer la manière dont les pointeurs sont employés. Lorsque l'on a compris comment ils fonctionnent avec un programme aussi simple que celui-ci, on comprend comment pourraient être ajoutés des pointeurs. Comme n'importe quel autre type de données, les pointeurs sont déclarés avec un identificateur et un type. Cependant, ils emploient une syntaxe inhabituelle. La section de déclaration de notre exemple apparaît ci-dessous :

```
TYPE
	Nom = STRING [17]
	PointeurEnregDetective = ^EnregDetective ;
	EnregDetective = RECCORD
		Detective : Nom ;
		DetectivePreced : PointeurEnregDetective ;
	END ;
CONST
	MaxElements = 5 ;
VAR
	Predecesseur, EnregDetectiveCourant, Temp :
	PointeurEnregDetective ;
	I : INTEGER ;
	UnNom : Nom ;
```

L'accent circonflexe (^) caractérise un pointeur ; le Turbo utilise également ce symbole pour représenter un caractère de contrôle. L'identificateur PointeurEnregDetective est défini comme un pointeur de EnregDetective qui est défini à son tour comme n'importe quel enregistrement. Les déclarations de pointeur sont les seules pour lesquelles le Turbo autorise l'emploi d'un terme avant sa déclaration. Dans cet exemple, ^EnregDetective est utilisé avant la déclaration de EnregDetective.

PointeurEnregDetective n'est qu'une adresse et ne contient pas les données de l'enregistrement. Le Turbo affiche un message d'erreur d'entrée/sortie si on lui demande d'afficher un pointeur sur l'écran. Pour afficher les données associées au pointeur, on utilise la commande suivante :

```
WRITELN (PointeurEnregDetective^.Detective) ;
```

Cette ligne affiche le nom du détective stocké dans l'enregistrement auquel le pointeur se réfère. La syntaxe est identique à celle qui est employée pour accéder aux éléments de données à l'intérieur d'enregistrements ordinaires.

L'identificateur standard NIL sert à signaler le dernier pointeur de la liste. Le programme commence par initialiser le premier pointeur à cette valeur. Les quatre lignes suivantes créent des pointeurs et leur assignent des valeurs :

```
NEW (EnregDetectiveCourant)
EnregDetectiveCourant ^.Detective : = UnNom ;
EnregDetectiveCourant ^.DetectivePreced : = Predecesseur ;
Predecesseur : = EnregDetectiveCourant ; {le predecesseur pointe
maintenant l'enregistrement en cours}
```

La procédure NEW alloue un espace du tas à l'enregistrement indiqué par le pointeur. A l'inverse du processus de dimensionnement de tableau, la procédure NEW n'est pas seulement exécutée une fois pour l'ensemble de la liste. Chaque combinaison pointeur/enregistrement supplémentaire est créée dynamiquement et individuellement chaque fois avec la procédure NEW. La Figure 4.38 montre plusieurs procédures pour attribuer des quantités d'espace spécifiques mais celles-ci sont employées dans des applications qui dépassent l'objet de cet ouvrage.

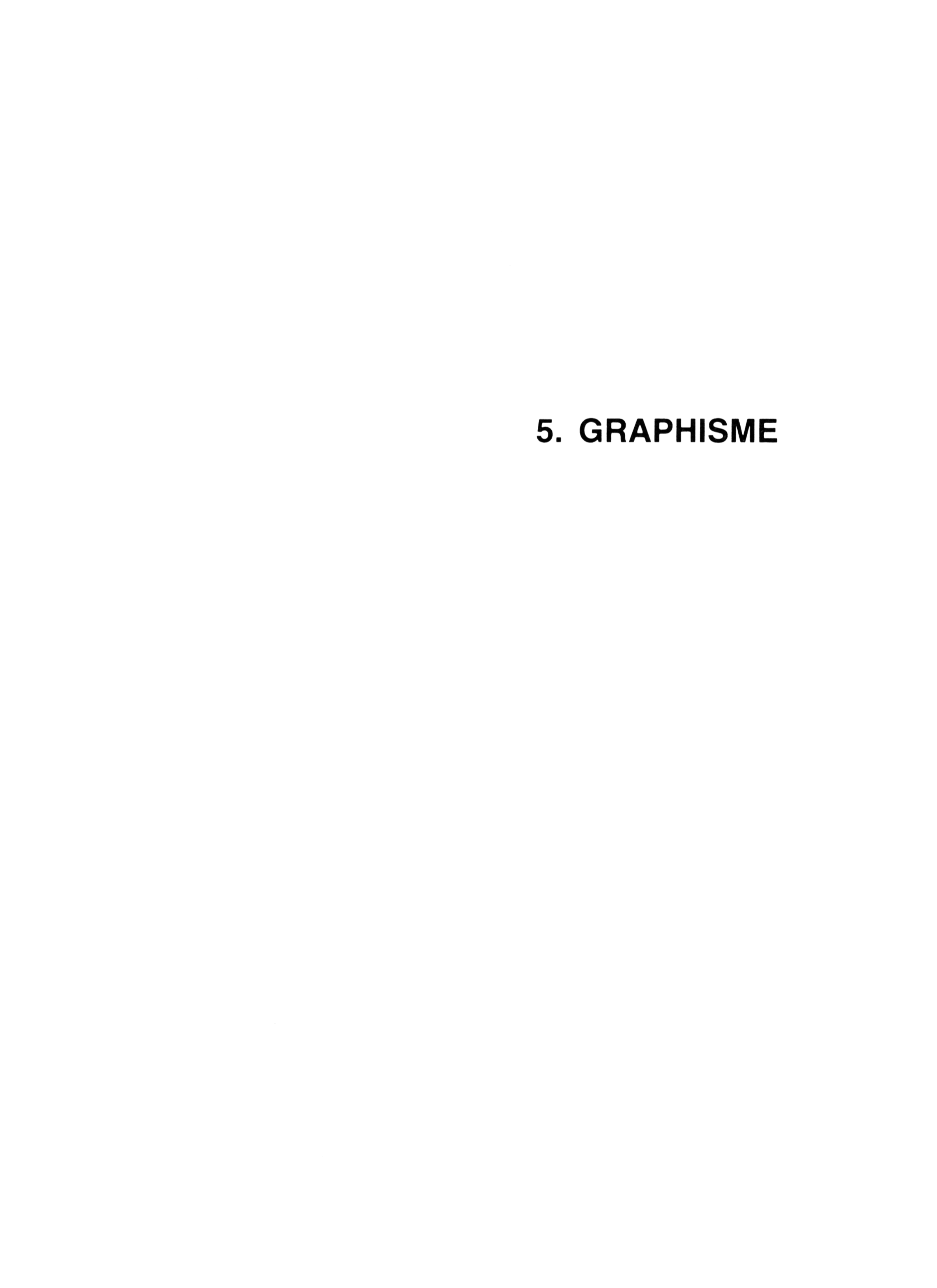
Après la création du pointeur par NEW, les données sont écrites dans l'enregistrement. Dans l'exemple de programme, un nom de détective est analysé dans la variable UnNom qui est ensuite assignée à EnregDetectiveCourant ^.Detective (dans un programme plus sophistiqué, l'enregistrement pourrait contenir plus de champs). La ligne de code suivante établit le pointeur à l'intérieur de l'enregistrement en cours pour pointer un autre enregistrement ; l'établissement de celui-ci est l'étape la plus délicate. Pour plus de simplicité, l'exemple charge l'adresse de l'enregistrement du prédécesseur dans ce pointeur ; de cette manière, chaque nouvel

enregistrement contient un pointeur pour l'enregistrement précédent. Le premier enregistrement contenant un nom de détective (Sherlock Holmes) possède un pointeur pour un enregistrement factice (prédécesseur) qui possède un pointeur pour NIL. Cet identificateur signale au Turbo la fin de la liste.

Si l'on avait employé des tampons et des pointeurs supplémentaires dans chaque enregistrement, ils auraient tous pu pointer leur successeur et leur prédécesseur, mais l'exemple de programme aurait été beaucoup trop complexe.

La procédure Sortie est initialisée avec le dernier enregistrement (Temp := EnregDetectiveCourant) et emploie une boucle, d'abord pour afficher le nom du détective, ensuite pour trouver l'enregistrement suivant. Elle utilise le test pour NIL comme pour EOF, de manière à continuer la lecture d'une liste de longueur inconnue.

Les listes et les pointeurs constituent un sujet qui mérite à lui seul de faire l'objet d'un ouvrage.



5. GRAPHISME

PRESENTATION

L'implémentation CP/M 80 du Turbo standard n'offre pas d'instructions graphiques ; le logiciel ne propose que quelques fonctions et procédures destinées à gérer l'affichage de texte sur l'écran. Ce chapitre traite du graphisme sur les ordinateurs Amstrad ; la première partie rappelle les fonctions et procédures permettant de contrôler l'affichage écran, la seconde partie présente la bibliothèque de fichiers intégrés appelée Graphix Toolbox.

CONTROLE DE L'AFFICHAGE ECRAN

Le Turbo offre plusieurs fonctions et procédures pour contrôler l'affichage écran ; elles sont particulièrement utiles pour rendre plus agréable l'interaction entre le programme et l'utilisateur. La Figure 5.1 montre les outils de manipulation d'écran disponibles dans la version standard du Turbo Pascal. La plupart de ces caractéristiques ont un nom très explicite.

Les abréviations utilisées dans la figure sont les suivantes :

I	Entier
V	Variable quelconque
Val	Caractère ou donnée de type octet.

Procédures et fonctions relatives à l'écran et au clavier

CLREOL	Efface tous les caractères depuis la position du curseur et jusqu'à la fin de la ligne sans déplacer le curseur.
CLRSCR	Efface la totalité de l'écran et place le curseur dans le coin supérieur gauche.
CRTINIT	Envoie à l'écran la chaîne d'initialisation de terminal (utilisée très rarement).
CRTEXIT	Envoie à l'écran la chaîne de réinitialisation de terminal (utilisée très rarement).
DELLINE	Supprime la ligne sur laquelle le curseur est placé et déplace les lignes en-dessous d'une ligne vers le haut.
FILLCHAR (V, I, VAL)	Remplit I octets de la mémoire à partir du premier octet occupé par V avec la valeur VAL (de type caractère ou de type octet).
GOTOXY (X, Y)	Place le curseur au point de coordonnées X, Y ; X est une position horizontale et Y est une position

Figure 5.1 : *Procédures et fonctions standard du Turbo Pascal pour la manipulation écran*

	verticale. Ce sont des valeurs entières mesurées à partir du coin supérieur gauche de l'écran.
HIGHVIDEO	Envoie à l'écran l'attribut vidéo HIGH comme défini dans la procédure d'installation de terminal.
INSLINE	Insère une ligne vide à la position du curseur.
LOWVIDEO	Envoie à l'écran l'attribut vidéo LOW comme défini dans la procédure d'installation de terminal.
MOVE (V1, V2, I)	Effectue une copie de la zone mémoire de I octets, commençant à partir du premier octet de la variable V1 ; cette copie est réalisée à partir du premier octet de la variable V2. Cette instruction était principalement utilisée pour des déplacements d'images graphiques à grande vitesse.
NORMVIDEO	Envoie à l'écran l'attribut vidéo NORM comme défini dans la procédure d'installation de terminal.

Figure 5.1 *(suite)*

GOTOXY a déjà été présentée comme une fonction permettant d'afficher un résultat à n'importe quel endroit de l'écran. NORMVIDEO, LOWVIDEO et HIGHVIDEO servent à faire apparaître une ligne de l'écran sous différentes formes (si ces caractéristiques sont disponibles sur l'ordinateur). Chaque mode vidéo reste effectif jusqu'à sa suppression par la frappe d'une autre commande. Les programmeurs chevronnés peuvent utiliser le programme d'installation de terminal pour définir la caractéristique associée à chaque commande ; il est même possible d'employer la fonction LOWVIDEO pour faire clignoter l'écran ou effacer tous les caractères.

DELLINE, INSLINE, CLREOL et CLRSCR opèrent exactement comme le décrit la Figure 5.1. De nombreux programmes de ce

livre emploient la procédure d'effacement d'écran CLRSCR. Les versions 1 et 2 du Turbo Pascal initialisent automatiquement l'écran au début de chaque programme. Après de nombreuses remarques des utilisateurs, le logiciel a été modifié et la version 3 ne réalise plus cet effacement systématique. Pour qu'un programme démarre sur un écran vierge (avec le curseur placé dans l'angle supérieur gauche), il faut que l'utilisateur insère cette procédure dans son programme.

CRTINIT et CRTEXIT sont en principe automatiquement exécutées par le Turbo et rarement employées dans un programme. Elles ont une fonction de soupape de sécurité dans la mise au point de programmes graphiques complexes (particulièrement dans les programmes qui peuvent diriger des résultats soit vers un moniteur monochrome, soit vers un moniteur couleur, pour initialiser le moniteur graphique sans avoir à lancer à nouveau l'ensemble du programme.

FILLCHAR et MOVE sont principalement utilisées en relation avec des techniques avancées d'adressage mémoire.

GRAPHISME SUR AMSTRAD AVEC GRAPHIX TOOLBOX

Graphix Toolbox est une bibliothèque de procédures et de fonctions qui donne au Turbo Pascal la possibilité d'accéder au graphisme. Elle permet de tracer des points, des lignes, des figures géométriques telles que des rectangles, des ellipses, des cercles, etc. ; du texte peut être inséré dans les figures, les courbes peuvent être tracées avec différents styles de traits. La taille des fichiers qu'elle comporte et l'espace mémoire occupé par les constantes et variables auxquelles elle fait appel rendent son usage impossible sur les ordinateurs CPC 464 et CPC 664.

Quatre fichiers intégrés constituent la base de Graphix Toolbox, ce sont :

```
TYPEDEF.SYS
GRAPHIX.SYS
KERNEL.SYS
KERNEL1.SYS
```

ils doivent être placés au début des programmes graphiques et dans l'ordre particulier ci-dessus.

Ces fichiers intégrés contiennent les déclarations de variables et de constantes ainsi que les définitions des fonctions et procédures constituant les commandes de Graphix Toolbox. Les fichiers intégrés sont traités en détail dans la sixième partie. Les fichiers TYPEDEF.SYS, GRAPHIX.SYS, KERNEL.SYS et KERNEL1.SYS doivent être présents sur le disque lors de la compilation du programme graphique. Une fois compilé, le programme d'application se suffit à lui-même et n'a plus besoin de fichiers externes.

La Figure 5.2 indique le rôle des différents fichiers intégrés de la bibliothèque graphique.

Les fichiers système de base :

Ils doivent être inclus dans toutes les applications graphiques du Turbo Pascal et placés dans l'ordre suivant :

TYPEDEF.SYS	Fichier de déclaration de constantes et de variables pour la bibliothèque Graphix Toolbox.
GRAPHIX.SYS	Fichier de déclaration de constantes, de variables, de procédures et de fonctions pour les tracés de base ainsi que pour le chargement et le stockage d'écrans graphiques.
KERNEL.SYS	Fichier de procédures et de fonctions pour l'initialisation et la gestion de la bibliothèque Graphix Toolbox.

Figure 5.2 : *Les fichiers de la bibliothèque Graphix Toolbox*

KERNEL1.SYS — Fichier de procédures et de fonctions pour la gestion de la bibliothèque Graphix Toolbox.

Fichiers système supplémentaires :

La taille mémoire disponible étant relativement restreinte sur le CPC 6128, il est conseillé d'utiliser des systèmes de chevauchement si l'on veut intégrer des fichiers système supplémentaires dans un programme. Les systèmes de chevauchement sont décrits dans le Chapitre 6.

WINDOWS.SYS — Fichier de procédures de gestion de fenêtres ; création, déplacement, chargement et sauvegarde.

4X6.FON — Fichier des polices de caractères Turbo Graphix.

ERROR.MSG — Fichier des messages d'erreur.

Les fichiers de commandes de haut niveau :

Pour éviter un dépassement de capacité à la compilation, ces fichiers doivent être compilés avec des systèmes de chevauchement (voir le chapitre 6).

FINDWRLD.HGH — Cette procédure détermine l'environnement graphique nécessaire au tracé d'un polygone ; adaptation automatique de l'échelle.

AXIS.HGH — Procédure de tracé d'axes et mise en place d'étiquettes.

Figure 5.2 *(suite)*

POLYGONHGH	Procédure de tracé de polygones.
MODPOLY.HGH	Procédure destinée à générer des rotations, des modifications de taille, des déplacements de polygones.
SPLINE.HGH	Procédure de lissage permettant l'interpolation de certains points lors du tracé d'un polygone.
BEZIER.HGH	Procédure de calcul de fonction de Bezier. Cette procédure est utilisée au cours du lissage d'une courbe pour générer une forme particulière.
HATCH.HGH	Procédure de remplissage d'une zone rectangulaire de l'écran avec des lignes obliques parallèles.
HISTOGRM.HGH	Procédure destinée à tracer des histogrammes.
CIRCSEGM.HGH	Procédure permettant de tracer des parties de cercles et de placer des étiquettes.
PIE.HGH	Procédure destinée à tracer des diagrammes sectoriels et à générer des zones de texte.

Figure 5.2 *(suite)*

Les fichiers intégrés sont insérés dans les programmes d'application grâce aux options de compilation :

```
{$I TYPEDEF.SYS}
{$I GRAPHIX.SYS}
{$I KERNEL.SYS}
{$I KERNEL1.SYS}
```

Le manuel de Graphics Toolbox propose quelques programmes graphiques puis la liste des commandes (procédures et fonctions). Chacune d'elles est suivie d'un nom placé entre crochets ; c'est le

nom du fichier dans lequel elle est définie. Si ce fichier est différent de l'un des quatre fichiers de base, il faut l'ajouter au programme.

Le programme de la Figure 5.3 trace le graphe de la fonction Y = EXP(aX) * SIN(X) + b dans un repère orthonormé (les axes X et Y sont perpendiculaires).

La première partie déclare les quatre fichiers à intégrer et définie la constante Pi.

La seconde partie définit la fonction Y(X).

La troisième partie trace les axes X et Y. La procédure DRAWSTRAIGHT dessine une ligne horizontale (y=100) du point de coordonnée x=19 au point de coordonnée x=619. La procédure DRAWLINECLIPPED trace une ligne du point de coordonnées x=319, y=9 au point de coordonnées x=319, y=189.

La quatrième partie trace les graduations sur les axes, la cinquième partie génère le dessin de la fonction et la sixième partie regroupe les procédure destinées au tracé de la figure.

Le corps principal du programme initialise l'écran graphique, trace la fonction sur l'écran puis attend qu'une touche soit frappée pour quitter l'écran graphique.

```
PROGRAM Sinusoide_amortie;


{$I TYPEDEF.SYS}            (* ces fichiers doivent etre
{$I GRAPHIX.SYS}               inseres dans cet ordre *)
{$I KERNEL.SYS}
{$I KERNEL1.SYS}
```

Figure 5.3 : *Tracé d'une fonction mathématique avec Graphix Toolbox*

```
CONST
   Pi = 3.1416;


FUNCTION Y(X : REAL) : REAL;
BEGIN
   Y := (EXP( 0.02 * X) * SIN (X) * 100) + 99;
END;


PROCEDURE Trace_axes;
BEGIN
   DRAWSTRAIGHT (19,619,100);
   DRAWLINECLIPPED (319,9,319,189);
END;


PROCEDURE Trace_gradu;
VAR
   I : INTEGER;

BEGIN
   I := 19;
   WHILE I <= 619 DO
      BEGIN
         DRAWLINECLIPPED (I,97,I,101);
         I := I + 20;
      END;
   I := 9;
   WHILE I <= 199 DO
      BEGIN
         DRAWSTRAIGHT (316,312,I);
         I := I + 20;
      END;
END;
```

Figure 5.3 *(suite)*

```
PROCEDURE Genere_fonction;
VAR
   X0, Y0 : INTEGER;
   X, I   : REAL;

BEGIN
   I := 0;
   X := 0;
   X0 := 19;
   Y0 := 100;
   WHILE I <= (30 * Pi) DO
      BEGIN
         X := X + 1;
         DRAWLINECLIPPED (X0,Y0,ROUND(X + 19),ROUND(Y(I));
         X0 := ROUND (X + 19);
         Y0 := ROUND (Y(I));
         I := I + (5 * Pi / 100);
      END;
END;

PROCEDURE Trace_fig;
BEGIN
   CLEARSCREEN;
   DRAWBORDER;
   Trace_axes;
   Trace_gradu;
   Genere_fonction;
END;

BEGIN
   INITGRAPHIC;                {initialise le systeme graphique}
   Trace_fig;
   REPEAT UNTIL KEYPRESSED;    {attend la frappe d'une touche}
   LEAVEGRAPHIC;               {quitte le systeme graphique}
END.
```

Figure 5.3 *(suite)*

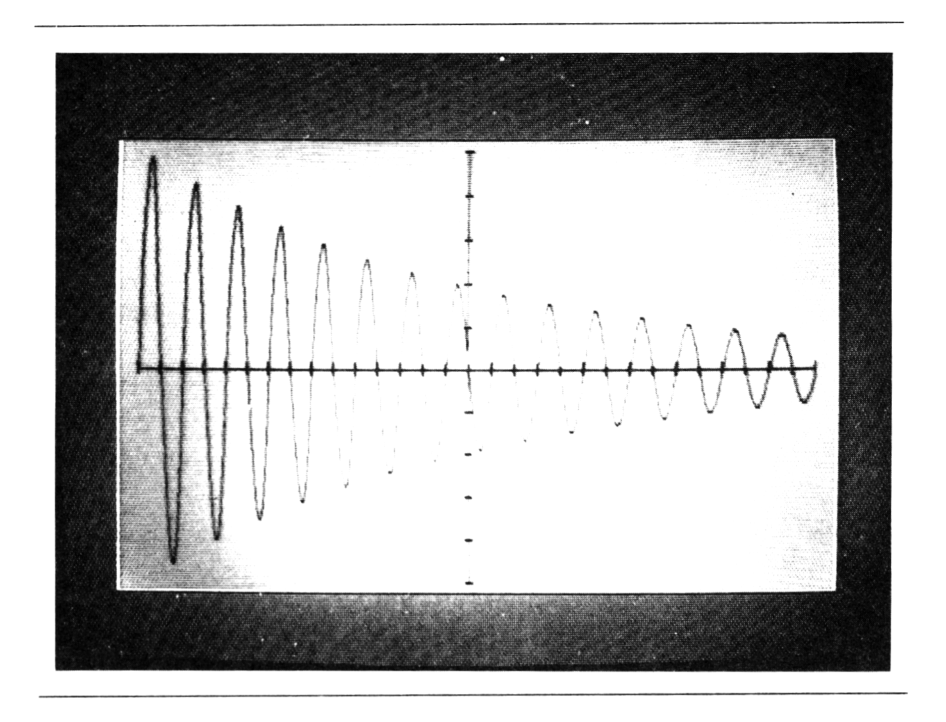


Figure 5.4 : *Tracé de la fonction sur l'écran*

6. PROGRAMMATION AVANCEE

OPTIONS DE COMPILATION

Chaque compilateur Pascal a ses caractéristiques propres et certaines opérations et options varient considérablement. Les chapitres du manuel de référence relatifs à l'installation du Turbo sont la seule source d'information concernant les opérations de compilation. Ceci est dommage car il existe plusieurs possibilités de mise au point puissantes et pratiquement inconnues alors qu'elles sont accessibles par un simple choix parmi diverses options. Certains programmeurs chevronnés ignorent ces options car les informations qui les concernent sont enterrées dans la documentation et doivent être glanées au hasard.

Le compilateur Turbo est le coeur du Turbo Pascal ; Borland a réussi à intégrer des instructions extrêmement puissantes dans un compilateur de 40k octets. Leur vitesse et leur capacité n'ont pas de précédent et sont déjà considérées comme un exemple de réalisation pour d'autres compilateurs. Borland a employé plusieurs moyens pour rendre le compilateur rapide. Nous avons déjà vu que les programmes sont entièrement écrits, exécutés et mis au point dans la mémoire centrale de l'ordinateur. Nous savons également que le Turbo a affiné le processus de compilation en ne générant pas de listing de compilation ni de table de références croisées. Une autre méthode pour améliorer la vitesse du compilateur (chaque version du Turbo semble plus rapide que la précédente) consiste à établir un jeu de directives de compilation choisies pour

une efficacité optimum. Pour la majorité des programmes susceptibles d'être écrits, le compilateur Turbo a des valeurs par défaut destinées à optimiser la vitesse. Normalement, il n'est pas nécessaire de sélectionner une option quelconque avant d'exécuter un programme dans l'environnement Turbo, il suffit de taper la touche R pour demander l'option compilation-exécution de l'éditeur Turbo. La constitution d'un fichier .COM nécessite seulement le choix de l'option C avant la compilation du programme.

Les nombreuses options de compilation sont choisies par l'utilisateur en fonction de ses besoins. La plupart de ces options sont des outils qui aident à la mise au point de programmes. Généralement, elles ralentissent le déroulement (parfois beaucoup) en vérifiant les erreurs. Lorsqu'un programme a été mis au point, il est possible de le recompiler en supprimant les options de compilation.

A l'inverse du Turbo, la plupart des compilateurs Pascal demandent que l'on réponde à une série de questions pour chaque compilation ; certains nécessitent également l'exécution de plusieurs processus séparés. Ceci est fastidieux surtout quand on sait que pour chaque symbole point-virgule (;) mal placé, il faut tout recommencer.

Les options de compilation Turbo se divisent en deux catégories principales : options de diagnostic, pour aider à la mise au point de programmes, et options destinées à répondre à des conditions inhabituelles. La Figure 6.1 donne la liste des directives disponibles à partir du compilateur ainsi que les valeurs par défaut de toutes les options. Valeur *par défaut* signifie valeur que le compilateur attribue à une directive si aucune indication contraire ne lui est donnée. Il ne faut pas oublier que Borland a choisi ces valeurs pour générer le programme le plus rapide ; par conséquent, il ne faut pas les modifier sans motif valable.

CODE	DEFAUT	DESCRIPTION
B	B+	Périphériques d'entrée/sortie. Sélectionne les périphériques logiques associés aux fichiers standard INPUT et OUTPUT ; B+ utilise CON: et B- utilise TRM:.
C	C+	Contrôle du clavier au cours de l'exécution d'un programme. C+ permet à la séquence Ctrl-C d'interrompre le programme et à la séquence Ctrl-S d'activer/désactiver les sorties sur l'écran. C- ignore ces séquences.
I	I+	Gestions des erreurs d'entrée/sortie. Contrôle les entrées/sorties et génère un message d'erreur en cas d'erreur. I- ignore les erreurs.
I	aucun	Fichiers externes inclus lors de l'exécution d'un programme. La directive I suivie d'un nom de fichier (par exemple, {$I LIBRAIRIE.PAS}) indique au compilateur d'associer ce fichier avec le fichier en cours lors de la compilation.
R	R-	Vérification des indices et scalaires. La directive R+ entraine la vérification des indices des tableaux ainsi que les affectations aux scalaires, par rapport aux limites déclarées.
U	U-	Interruption au cours de l'exécution d'un programme. U+ permet d'interrompre le programme lors de la frappe de la séquence Ctrl-C. A l'inverse de la directive C, elle n'a aucun effet sur la séquence Ctrl-S.

Figure 6.1 : *Directives de compilation et options disponibles dans toutes les versions du Turbo Pascal*

V	V-	Vérification de la longueur des chaînes lors du passage de paramètres. Vérifie que les variables de chaînes passées comme paramètre ont une longueur identique à celles de la procédure appelée.

Figure 6.1 *(suite)*

Toutes les options de compilation sont établies par des directives, lignes de commentaireS à l'intérieur des programmes dont la syntaxe particulière indique au Turbo qu'il ne s'agit pas réellement de commentaires mais plutôt de commandes destinées au compilateur. Les directives commencent toutes par le signe dollar ($) ; la plupart d'entre elles sont des codes à un caractère avec un signe plus (+) pour autoriser une option et un signe moins (-) pour la refuser. De plus, elles peuvent presque toutes être activées ou désactivées dans le programme. Cependant, quelques-unes (telles que B, C et U) doivent s'appliquer à l'ensemble du programme ; ce point n'est pas très clair dans la documentation. De plus, si elles ne sont pas invoquées dans la première ligne suivant l'en-tête du programme, ces options fonctionnent de manière hypothétique ou même pas du tout.

Plusieurs options de compilation méritent des explications supplémentaires.

Périphériques d'entrée/sortie par défaut (B). En principe, les fichiers standard INPUT et OUTPUT sont associés au périphérique logique console (CON:), avec mémoire tampon et écho. Lorsqu'elle est désactivée (B), ces fichiers sont associés au périphérique logique terminal (TRM:). Il ne faut pas oublier que le périphérique terminal ne stocke pas les entrées dans une mémoire tampon et supprime l'écho écran de la plupart des caractères de contrôle (mais pas des caractères texte). Cette option ne peut pas être désactivée à l'intérieur d'un programme.

Contrôle du clavier durant l'exécution du programme (C et U). Pendant l'exécution d'un programme, la caractéristique par défaut

de l'option C (C+) permet de suspendre l'affichage écran par l'intermédiaire de la frappe des touches Ctrl-S puis de le reprendre par une nouvelle frappe de ces touches. Cette option est utile lorsqu'un programme génère un affichage trop rapide pour que l'on puisse le lire. De même, la frappe de Ctrl-C met fin au déroulement du programme, ce qui permet de s'arrêter dans une boucle ou d'interrompre un affichage long lorsqu'une information recherchée est trouvée. Si cette option est désactivée (C), Ctrl-C et Ctrl-S n'ont aucun effet sur le programme. Cette option est activée par défaut. L'option U n'a aucun impact sur l'opération Ctrl-S mais lorsqu'elle est activée (U+), la frappe des touches Ctrl-C interrompt le programme. La condition par défaut de C autorise l'utilisation de Ctrl-C alors que la condition par défaut de U ne l'autorise pas. En utilisant simultanément les deux options, on peut obtenir trois conditions d'opérations différentes :

C+, U- : condition par défaut, Ctrl-S et Ctrl-C contrôlent le programme

C-, U- : Ctrl-S et Ctrl-C n'ont aucun effet

C-, U+ : Ctrl-S n'a aucun effet mais Ctrl-C interrompt le programme.

Lorsqu'elles ont été établies, U et C ne peuvent pas être modifiées à l'intérieur d'un programme.

Vérification d'erreur d'entrée/sortie (I+ et I-). Toutes les opérations d'entrée/sortie sont vérifiées par le Turbo. Lorsqu'une erreur est rencontrée, le programme est interrompu et le message d'erreur approprié est affiché. En programmation avancée, on peut désactiver (I-) cette vérification automatique pour que le programme ne s'arrête pas en cas d'erreur. Lorsque la désactivation est sélectionnée, le programme doit contenir un sous-programme de gestion d'erreur (utilisant la fonction standard IORESULT) pour analyser les erreurs et demander à l'utilisateur de les corriger. Si ce sous-programme n'existe pas, on risque d'obtenir un déroulement imprévisible et de revenir au système d'exploitation. Dans les programmes écrits par des programmeurs chevronnés, on voit parfois une désactivation de vérification d'erreurs concernant quelques lignes de code puis une activation pour le reste du programme. Cette technique est pratique pour éviter certaines

erreurs d'entrée/sortie classiques mais il faut la maîtriser avant de s'en servir.

Insertion d'un fichier externe pendant l'exécution d'un programme (I avec un nom de fichier). La directive I à l'intérieur d'un programme (appelé *programme de travail* par l'éditeur Turbo) indique au compilateur d'ouvrir le fichier spécifié et d'insérer son contenu dans la version compilée du programme de travail. Une fois ce programme compilé, il comprend chaque ligne du fichier inclus et n'a plus besoin de ce fichier pour les opérations ultérieures. Le nombre de fichiers à inclure dans le programme d'appel est illimité. Par conséquent, cette méthode donne au programmeur le moyen d'accéder à plusieurs bibliothèques de procédures et de fonctions pour des applications particulières.

Il y a deux "bugs" dans la directive d'inclusion de fichiers. Le programme TLIST affiche un message d'erreur et non le contenu d'un fichier inclus si on emploie la syntaxe standard :

```
{$I BIBLIOTHEQUE.PAS}
```

Cependant, le problème disparaît et TLIST affiche le fichier principal et les fichiers inclus si on omet l'espace placé avant le nom de fichier :

```
{$IBIBLIOTHEQUE.PAS}
```

Dans certaines versions du Turbo, pour un fonctionnement efficace, le nom du fichier à inclure doit être composé d'au moins 8 caractères ou doit avoir une extension. Il est impossible d'inclure un fichier à l'aide de la syntaxe suivante :

```
{$I TAB}
```

Cependant, avec une extension, ce nom est valide :

```
{$I TAB.XXX}
```

En principe, les fichiers ne sont que des collections de déclarations, de procédures et de fonctions et ne peuvent pas

constituer de programmes à part entière. Si on essaie d'inclure un fichier entier dans un autre fichier, le Turbo voit deux couples BEGIN/END et génère un message d'erreur. Par conséquent, les fichiers inclus ne sont jamais compilés séparément mais seulement stockés sur disque. Pour aider à la mise au point, la plupart de ces fichiers contiennent un petit programme qui appelle les fonctions et les procédures pour les tester. Une fois les fonctions testées, il est supprimé de telle manière que le compilateur ne le voie pas lorsque le fichier est inclus ; il peut facilement être réinséré si un test ultérieur est nécessaire. Dans le programme de la Figure 6.11, les commentaires indiquent les lignes à enlever de ce fichier pour convertir ainsi un programme de travail en un fichier à inclure.

Vérification des indices (R). Lorsque cette option est activée (R+), le Turbo vérifie tous les indices de tableaux pour s'assurer qu'ils font partie d'une gamme de valeurs déclarées. Il vérifie également les valeurs assignées aux scalaires pour s'assurer qu'elles sont dans les limites standard définies par l'utilisateur. La vérification des indices ralentit l'exécution du programme et Borland a choisi comme condition par défaut de désactiver cette option. Les programmeurs chevronnés activent parfois la vérification des indices pendant la mise au point de programmes et la désactivent pour une exécution plus rapide du produit fini. Les programmeurs occasionnels trouvent que même avec la valeur par défaut de l'option, le Turbo effectue suffisamment de vérifications d'erreurs pour répondre à leurs besoins.

Vérification du type pour les paramètres variables (V). Lorsque cette option est activée, le Turbo vérifie si les chaînes passées comme des paramètres de variables (VAR) ont exactement la même longueur que celles qui sont définies à l'intérieur de la procédure appelée. Lorsqu'elle est désactivée, le Turbo autorise le passage de chaînes de longueur quelconque dans la procédure mais les résultats sont imprévisibles si la chaîne est trop longue pour la procédure cible. On peut minimiser les problèmes potentiels en écrivant des procédures cible de telle sorte qu'elles puissent manipuler la plus longue chaîne qu'elles risquent de devoir traiter (255 caractères maximum, 80 caractères pour les chaînes basées sur les entrées de lignes, 14 pour les chaînes qui contiennent les identificateurs de fichiers MS-DOS, etc.).

Il existe des directives de compilation supplémentaires qui dépendent du système d'exploitation. Elle sont principalement intéressantes pour l'écriture de programmes qui utilisent des types d'entrées/sorties non standard. Elles apparaissent dans la Figure 6.2 pour une simple référence.

CODE	DEFAUT	DESCRIPTION
A	A+	Code absolu. Permet de générer un code absolu non récursif.
W	W2	Commandes WITH imbriquées. Indique le nombre maximum d'instructions WITH pouvant être imbriquées.
X	X+	Optimisation des tableaux. Lorsqu'un tableau est généré, le code est optimisé en fonction de la vitesse d'exécution (X+) ou en fonction de la place mémoire occupée (X-).

Figure 6.2 : *Directives et options de compilation spécifiques au système d'exploitation CP/M-80*

LA PROCEDURE EXECUTE

Le Turbo offre deux autres techniques d'interaction entre des fichiers. La procédure EXECUTE donne à un programme Turbo l'autorisation d'exécuter un autre programme. Cependant, il est impossible d'exécuter un programme qui n'a pas été créé par le Turbo Pascal. Il ne faut pas confondre la procédure EXECUTE avec le concept de fichier inclus. Lorsque l'on exécute un programme à partir d'un autre programme, le programme initial est remplacé en mémoire par le second programme. Les deux fichiers restent indépendants. Lorsque le second programme a réalisé

l'exécution, il ne revient pas au premier programme qui n'existe plus en mémoire.

Pour illustrer le fonctionnement de la procédure EXECUTE, on compile le programme de la Figure 5.3 de manière à créer un fichier .COM. Dans cet exemple, la tâche à exécuter s'appelle SIN_A.COM. Nous allons modifier le programme de tri de Shell de la Figure 4.1 pour qu'il affiche sur l'écran une sinusoïde amortie à la fin du tri. Ce processus permet par une modification notable de l'écran de signaler à l'utilisateur qu'une tâche longue est terminée.

La procédure EXECUTE permet, par exemple, d'ajouter un programme de spooling qui génère l'impression d'une liste d'adresses sur des étiquettes adhésives à un programme de tri par ordre alphabétique. Ce type de programme dont la caractéristique est de travailler en temps partagé est conçu pour contrôler la sortie sur un périphérique relativement lent et continuer l'exécution d'un programme principal. Initialement destinés à stocker des données dans une mémoire tampon afin de ne pas ralentir la vitesse de travail de l'ordinateur (puisqu'il doit s'adapter aux caractéristiques de périphériques mécaniques) les programmes de ce type sont maintenant couramment employés pour exploiter les contrôles de format des imprimantes modernes. TLIST, servant à l'impression des listings Turbo, en est un exemple classique ; il contrôle l'espacement de ligne, les marges, le soulignement et la numérotation de ligne et de page.

On modifie le programme de la Figure 4.1 pour inclure les trois nouvelles lignes de la Figure 6.3.

Il faut connaître le nom du programme à exécuter et il doit s'agir d'un fichier .COM. Le nom de fichier est assigné à une variable de fichier à l'aide de la procédure ASSIGN habituelle. La clarté et la simplicité de la syntaxe de la procédure EXECUTE sont illustrées dans la Figure 6.3. Il faut déclarer une variable du type FILE, DessFich dans cet exemple. Ensuite, on doit assigner le nom du fichier que l'on veut exécuter, SIN_A.COM dans cet exemple, à la variable de fichier et, enfin, associer la variable de fichier à l'instruction EXECUTE. Il est impossible de tester l'exécution d'un programme par un autre programme lorsque l'on se trouve dans l'environnement Turbo. Le programme initial doit également être compilé et exécuté en dehors du Turbo pour que la

procédure EXECUTE fonctionne. Si on compile le programme de la Figure 6.3 et qu'on l'exécute à partir du système d'exploitation, il affiche tous les messages du programme de tri de Shell et il dessine automatiquement la sinusoïde amortie. Lorsque la figure est terminée, le programme revient au système d'exploitation ; son signal apparaît. Avec un peu d'imagination, on peut créer des choses amusantes à l'aide de programmes qui exécutent d'autres programmes.

```
PROGRAM Tri_de_Shell;          {tri de Shell sur 500 nombres aléatoires}

CONST
   MaxEle = 500;

VAR
   Nums : ARRAY [1..500] OF INTEGER;
   Temp, I, J, Pass, Vide : INTEGER;
   DessFich : File;            {cette ligne est ajoutee}

.
(La partie centrale du programme n'a pas ete modifiee)
.

BEGIN
   WRITELN ('Début du programme de tri de Shell.');
   Entree_Gen;
   WRITELN ('Le tableau de ',MaxEle,' éléments est chargé et le tri
commence.');
   Tri;
   WRITELN ('Le tri est terminé ; il y a ',MaxEle,' valeurs classées.');
   Sortie
   ASSIGN (DessFich, 'SIN_A.COM');      {cette ligne est ajoutee}
   EXECUTE (DessFich);                  {cette ligne est ajoutee}
END.
```

Figure 6.3 : *Modification du programme de tri de Shell de la Figure 4.1 pour exécuter (EXECUTE) le programme appelé SIN_A.COM*

LA PROCEDURE CHAIN

La procédure CHAIN est plus complexe que la procédure EXECUTE et que les processus d'inclusion de fichiers. On a remarqué que même les plus petits programmes Turbo dépassent 9000 octets lorsqu'ils sont compilés ; d'autre part, les fichiers .COM s'accroissent moins vite que les fichiers .PAS parce que le compilateur Turbo inclut automatiquement la bibliothèque d'exécution. Celle-ci comprend un message de Copyright et la définition des identificateurs et des procédures standard. Cependant, lorsque ces quelque 9000 octets d'information font partie du fichier compilé, chaque procédure supplémentaire écrite n'ajoute que quelques octets de code compilé au programme exécutable.

La procédure CHAIN indique au Turbo de compiler le programme sans inclure la bibliothèque d'exécution ; cependant, sans la bibliothèque, le fichier chaîné ne peut pas se dérouler. Par contre, lorsqu'il est appelé à l'aide de la procédure CHAIN par un programme qui contient la bibliothèque, il tourne parfaitement.

La notion de chaînage est une variante de la notion de fichier inclus. Certaines personnes considèrent le chaînage comme étant une étape intermédiaire entre un fichier inclus qui ne peut pas être exécuté indépendamment ni être compilé et les fichiers .COM totalement indépendants (comme le fichier SIN_A.COM de l'exemple) qui sont exécutables séparément.

En Turbo, le chaînage n'est pas toujours un processus automatique et il faut s'assurer que la quantité de mémoire allouée au programme d'appel est suffisante pour le fichier chaîné. Si le programme chaîné est moins important que le programme d'appel, le processus est automatique. Pour illustrer cela, on recompile le programme "Sinusoïde_amortie" de la Figure 5.3 mais au lieu de choisir l'option de menu C (Compilation), on sélectionne l'option H (cHin). En utilisant le fichier source appelé SIN_A.PAS, cette opération génère un fichier appelé SIN_A.CHN. On revient au programme de tri de Shell modifié et on change les deux lignes suivantes :

```
ASSIGN (DessFich, 'SIN_A.COM'); {Cette ligne est ajoutee}
EXECUTE (DessFich);             {Cette ligne est ajoutee}
```

pour utiliser des fichiers de chaînage et la procédure CHAIN au lieu de la procédure EXECUTE :

```
ASSIGN (DessFich, 'SIN_A.CHN'); {Utiliser  .CHN et non .COM}
CHAIN (DessFich);
```

On compile le programme de tri de Shell à l'aide de l'option de menu C pour générer un fichier .COM puis on sort du Turbo et on exécute le programme. Il doit effectuer le tri puis afficher le dessin sur l'écran.

Jusque-là, les opérations EXECUTE et CHAIN semblent pratiquement identiques. En quoi se différencient-elles ? La Figure 6.4 est un fragment de répertoire montrant la taille des trois versions différentes du fichier programme de la Figure 5.3. On voit que le fichier source ASCII (.PAS) comprenant tous les identificateurs et les commentaires occupe 386 octets, que le fichier exécutable (.COM) comprenant la bibliothèque d'exécution du Turbo occupe 16472 octets et que le fichier de chaînage (.CHN) est plus petit avec 11746 octets. Il ne faut pas oublier que le fichier exécutable est un fichier autonome qui peut se dérouler en dehors de l'environnement Turbo ; le fichier de chaînage opère également en dehors du Turbo lorsqu'il est associé à un fichier .COM. Le fichier SHELL.COM (nom du programme de tri de Shell compilé) occupe 15278 octets, il utilise la procédure EXECUTE ou la procédure CHAIN.

```
SIN_A   PAS     386
SIN_A   CHN     11746
SIN_A   COM     16472
SHELL   COM     15278
```

Figure 6.4 : *Extrait de répertoire montrant la longueur des fichiers source, de chaînage et d'exécution du programme de la Figure 5.3*

Même pour un programme tel que SIN_A, la différence d'occupation mémoire d'un format exécutable et d'un format chaînable peut avoir des conséquences surtout si le programme principal est important et dans les applications qui requièrent une vaste bibliothèque de modules.

Le processus de chaînage demande une attention particulière. Comme avec la procédure EXECUTE, le programme appelé prend en mémoire la place du programme d'appel. Ce processus se fait en deux temps ; les parties codées du programme (instructions) appelé sont chargées (la bibliothèque d'exécution du Turbo reste en mémoire). De même pour les parties du programme contenant les données (stockage des variables), à l'exception des variables communes aux deux programmes. Il est possible de passer des variables du programme initial à celles du programme chaîné et inversement. Pour cela, les deux programmes doivent être compilés. D'autre part, les variables communes aux deux programmes doivent être déclarées dans chacun d'entre eux, dans un ordre identique et avant les variables locales. Quand il accède au second programme, le Turbo initialise automatiquement les variables s'il rencontre deux variables portant le même nom (déclarées dans les deux programmes) à la même place en mémoire. Par conséquent, toute valeur stockée dans une variable par le premier programme est gardée en mémoire pour être utilisée par le second. Si le programme initial est plus important que le programme chaîné, il n'y a pas de problème. S'il est plus important que le programme d'appel, le programme chaîné n'est pas exécuté. Après la frappe de la touche O pour sélectionner le menu d'option, l'ordinateur affiche le menu de la Figure 6.5.

```
compile -> Memory
           Com-file
           cHn-file

Find run-time error Quit
```

Figure 6.5 : *Menu de sélection du mode de compilation*

Après la frappe de la touche H, sélection des fichiers .CHN, le menu propose à l'utilisateur de modifier l'adresse de départ ou l'adresse de fin (voir la Figure 6.6). La compilation écrit le code dans le fichier SIN_A.CHN, il contient le code du programme mais ne contient pas la librairie du Turbo Pascal.

```
           Memory
           Com-file
compile -> cHn-file

Start adress: 20E2 (min 20E2)
End   adress: 0000 (max F606)

Find run-time error Quit
```

Figure 6.6 : *Menu après sélection du mode de compilation cHn*

Si le code de chaînage (SIN_A.CHN) avait été plus important que le code SHELL, on aurait pu, lors de la compilation de SHELL, modifier la valeur de départ afin d'augmenter la zone occupée par son code pour réserver une place suffisante au chargement du programme SIN_A.CHN. A chaque sélection de l'option C dans le menu, le Turbo permet de modifier ces valeurs. Si on tape S ou E, le Turbo demande une nouvelle valeur ; la valeur entrée pour l'adresse de départ doit être supérieure à la valeur par défaut de la bibliothèque d'exécution (20E2 en hexadécimal). Elle doit être au moins égale à 20E2 plus la différence entre la taille du fichier initial et la taille du fichier de chaînage. La détermination de cette valeur nécessite des calculs fastidieux. La Figure 6.7 illustre la modification des valeurs de départ et de fin.

```
           Memory
compile -> Com-file
           cHn-file

Start adress: 4000 (min 20E2)
End   adress: 0000 (max F606)

Find run-time error Quit

End adress: F000
```

Figure 6.7 : *Options affichées par le compilateur Turbo*

SYSTEME DE RECOUVREMENT

Le Turbo (excepté la version 1) propose une autre technique de déplacement de programmes compilés. Il y a quelques années, la mémoire coûtait beaucoup plus cher que le "temps programmeur". En conséquence, les programmeurs consacraient beaucoup de temps à la "compression" de programmes complexes pour qu'ils n'occupent que 4, 8 ou 16 k de mémoire. Pour que des applications puissent tenir dans un espace aussi limité, il fallait constamment transférer des parties de programmes entre le disque (ou les bandes magnétiques) et la mémoire. Les sytèmes de recouvrement sont des parties de programme échangées entre le disque et la mémoire. Récemment encore, l'exploitation de cette capacité demandait une programmation fastidieuse et astucieuse. Le programmeur devait savoir exactement quelles étaient les parties du programme en mémoire à un moment quelconque et prêter attention à tous les détails du programme ainsi qu'à la taille réservée au recouvrement pour empêcher un chevauchement accidentel. Les calculs étaient beaucoup plus complexes que ceux qui servent à s'assurer que le fichier .COM alloue suffisamment d'espace au fichier .CHN.

Maintenant, la plupart des ordinateurs personnels disposent d'au moins 64 k de mémoire ; les systèmes de chevauchement sont bien moins nécessaires. Avec le Turbo, il n'est même pas indispensable d'envisager l'emploi de systèmes de chevauchement tant que la taille des programmes compilés (.COM) n'excède pas 64 k. Comme nous l'avons vu précédemment, une fois l'espace attribué à la bibliothèque d'exécution du Turbo, le compilateur est très peu "vorace" en mémoire car il convertit des milliers d'octets de code source en centaines d'octets de code exécutable.

Néanmoins, le Turbo propose un système de chevauchement pour les programmes très importants ; il ne demande pas au programmeur de calculer une attribution d'espace pour les chevauchements car il exécute automatiquement leur traitement.

Dans le fichier source .PAS, il suffit d'insérer le mot réservé OVERLAY avant le mot réservé PROCEDURE pour chaque procédure à stocker dans la zone de chevauchement, en dehors du fichier principal. Il est conseillé cependant de garder présentes en mémoire les procédures couramment utilisées et de reléguer les autres dans le fichier de chevauchement à partir duquel elles seront chargées en mémoire si nécessaire.

On peut ajouter un nombre illimité de procédures déclarées consécutivement dans le même fichier de chevauchement. Si un programme source contenait dix procédures, on pourrait placer les deux premières et les deux dernières dans des fichiers de chevauchement et les autres en mémoire. Le Turbo placerait chaque groupe de procédures consécutives dans des zones de chevauchement séparées. Par conséquent, les deux premières formeraient le fichier de chevauchement .000 et les deux dernières le fichier .001. Lors de la compilation du programme, le Turbo crée automatiquement le (ou les) fichiers de chevauchement. Il porte le même nom que le programme principal mais avec une extension de fichier séquentiel.

La Figure 6.8 est un répertoire montrant un programme compilé avec et sans système de chevauchement. Le fichier appelé MAIL.PAS est le fichier source d'un vaste fichier d'adresses similaire à celui de l'Annexe ; Il occupe 13468 octets. Lorsqu'il est compilé avec les systèmes de chevauchement, il génère un fichier .COM de 17598 octets. Les sytèmes de chevauchement ne sont pas indispensables car ce fichier ne dépasse pas les 64 k autorisés.

Cependant, pour illustrer leur emploi, le fichier source a été copié dans un fichier appelé MAILOVER.PAS puis modifié pour utiliser les cinq procédures du tri alphabétique dans un système de chevauchement ; ce nouveau fichier occupe 13508 octets. La compilation de MAILOVER a généré 2 fichiers de sortie : MAILOVER.COM et MAILOVER.000, le deuxième étant le fichier de chevauchement. En comparant la taille des deux fichiers .COM, on voit que l'utilisation de fichiers de chevauchement a permis de réduire la taille du fichier MAIL.COM de 1052 octets. Ces octets ainsi que les octets supplémentaires requis par le Turbo pour traiter les systèmes de chevauchement apparaissent dans le fichier MAILOVER.000.

```
Mail     PAS   13468
MAIL     COM   17598
MAILOVER PAS   13508
MAILOVER COM   16546
MAILOVER 000   1536
```

Figure 6.8 : *Extrait de répertoire montrant les fichiers de chevauchement*

Du point de vue de l'utilisateur, MAIL et MAILOVER opèrent de façon identique. Les programmes importants peuvent être quelque peu ralentis pendant que le Turbo lit les fichiers de chevauchement à partir du disque. Cependant, ces délais sont minimisés en plaçant uniquement les procédures les moins souvent employées dans les zones de chevauchement et en utilisant des disques virtuels et des disques durs. Le Turbo autorise la création de fichiers de chevauchement multiples et s'occupe alors d'accéder aux fichiers appropriés quand cela est nécessaire.

ACCES A DES COMMANDES DU SYSTEME D'EXPLOITATION

La gamme d'outils et de techniques abordée jusqu'à présent suffit à gérer un grand nombre de tâches de programmation. L'extraordinaire rapidité du Turbo et son jeu de structures de contrôle, de structures de données et de fonctions personnalisées offrent des possibilités qui dépassent celles de la plupart des autres langages.

Les 5 premiers chapitres de ce livre n'ont couvert que les 2/3 des capacités du Turbo. Ils ont abordé des éléments fondamentaux tels que les types de données, les structures de contrôle et la manipulation de ces structures de données dont la connaissance est indispensable avant de pouvoir exécuter des tâches plus complexes. De plus, la plupart de ces tâches nécessitent une connaissance plus approfondie du système d'exploitation de l'ordinateur. Des sujets tels que l'adressage et la manipulation directs de données stockées en mémoire, les appels au BIOS et au BDOS, la gestion des interruptions, l'adressage de ports et la manipulation de bits requièrent tous une connaissance spécifique de l'ordinateur et une inclination vers l'expérimentation.

Pour les nombreux programmeurs qui portent les stigmates de leur expérience de programmation en langage machine, le Turbo offre la rapidité et la puissance sans les obliger à plonger dans la circuiterie interne de l'ordinateur.

Il est impossible de donner dans ce livre autre chose qu'un catalogue de ces caractéristiques particulières. Nous allons cependant aborder brièvement les outils les plus utiles du système d'exploitation.

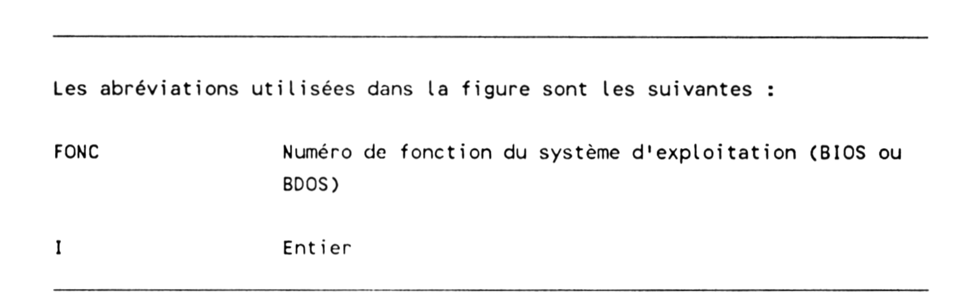
BIOS et BDOS

Le système d'exploitation CP/M possède une collection de commandes pour exécuter une large gamme de fonctions élémentaires toutes accessibles aux programmeurs en langage assembleur ; elles sont maintenant également accessibles aux programmeurs Turbo. Ces commandes se divisent en deux catégories : BIOS (système d'entrée/sortie de base) et BDOS

(système d'exploitation de base du disque). Il n'y a pas de séparation stricte entre les deux ; elles contrôlent (et donnent accès à) chacune des facettes des opérations de l'ordinateur, y compris les accès disque, CPU et ports de communication ainsi que la gestion d'erreur, le temps partagé et la gestion d'écran.

Pour exploiter ces possibilités, il faut posséder la documentation vendue par Digital Research (CP/M Plus Handbook ; Operator's and Programmer's Guide for the Amstrad CPC6128 and PCW8256) et spécifique au système d'exploitation de l'ordinateur ; il faut aussi une bonne compréhension de son fonctionnement interne. Si des termes tels que *registre*, *symbole d'état indicateur*, *table d'allocation de fichiers*, *interruption* et *vecteur* n'évoquent rien pour le lecteur, il peut s'arrêter là.

La Figure 6.9 résume l'assortiment de procédures disponibles avec CP/M. On crée un enregistrement composé de variables qui contiennent les valeurs stockées dans chacun des registres de l'ordinateur. Dans certains cas, on charge des valeurs critiques dans des registres particuliers. Dans d'autres cas, on lit simplement les valeurs retournées dans des registres ; parfois on cumule les deux opérations. Généralement, l'enregistrement est passé comme un paramètre avec un entier qui se réfère à une adresse mémoire ou un numéro d'appel de fonction. La Figure 6.9 donne la syntaxe spécifique à chaque procédure ainsi que sa définition.


Les abréviations utilisées dans la figure sont les suivantes :

FONC	Numéro de fonction du système d'exploitation (BIOS ou BDOS)
I	Entier

Figure 6.9 : *Procédures et fonctions standard en Turbo Pascal pour accéder aux fonctions du système d'exploitation CP/M*

X	Variable particulière décrite dans le descriptif associé.

Fonctions et procédures spécifiques à CP/M

BDOS (FONC, I)	Procédure permettant d'appeler la fonction BDOS de CP/M de numéro FONC ; I est un entier facultatif qui est chargé dans le registre DE. Un appel à l'adresse 5 exécute ensuite la fonction BDOS.
BDOS	Fonction retournant la valeur entière chargée dans le registre A par le BDOS de CP/M à la suite de l'exécution d'une procédure BDOS ou BDOSHL.
BDOSHL (FONC, I)	Procédure permettant d'appeler la fonction BDOS de CP/M de numéro FONC ; I est un entier facultatif qui est chargé dans le registre HL. Un appel à l'adresse 5 exécute ensuite la fonction BDOS.
BIOS (FONC, I)	Procédure permettant d'appeler la fonction BIOS de CP/M de numéro FONC ; I est un entier facultatif qui est chargé dans le registre BC.
BIOS	Fonction retournant la valeur entière chargée dans le registre A par le BIOS de CP/M à la suite de l'exécution d'une procédure BIOS ou BIOSHL.
BIOSHL (FONC, I)	Procédure permettant d'appeler la fonction BIOS de CP/M de numéro FONC ; I est un entier facultatif qui est chargé dans le registre HL.

Figure 6.9 *(suite)*

Le Turbo ne propose pas de commandes similaires aux commandes DATE$ et TIME$ du BASIC Microsoft pour aller chercher et afficher la date et l'heure ou bien la fonction TIME du

BASIC Locomotive Software qui renvoie le temps écoulé depuis la mise sous tension ou la dernière réinitialisation. Cependant, CP/M possède des fonctions qui peuvent générer ces informations. La Figure 6.10 montre un programme largement commenté qui crée une fonction appelée DATE. Cette fonction "va chercher la date" avec une fonction CP/M et la convertit à partir du format interne du DOS en une forme facilement affichée et manipulée par un programme Turbo.

La commande CP/M DATE est destinée à afficher ou modifier (sous DOS) la date et l'heure. Lorsque le système d'exploitation a été chargé, il est possible d'établir la date et l'heure en donnant à la commande le format suivant :

```
DATE <mm-jj-aa> <hh:mm:ss>
```

Il est indispensable d'avoir exécuté cette commande avant le chargement du Turbo Pascal si l'on veut donner un sens au programme de la Figure 6.10.

```
PROGRAM FonctionDate;

TYPE
   ChaineDate = STRING[40];

FUNCTION CDate : ChaineDate;  {retourne la date décodée}

TYPE
     Date = INTEGER;
     Adresse = ^Date;
```

Figure 6.10 : *Programme qui recherche, décode et affiche la date à l'aide d'une fonction BDOS du système d'exploitation*

```
VAR
   RegC : BYTE;
   LitDate : Adresse; {contient les valeurs renvoyées par le système}
   Donnee : INTEGER; {contient les valeurs renvoyées par le système}
   Mois : STRING[8];
   Jour : STRING[2];
   Annee : STRING[4];
   JourS : STRING[8];
   Annee1, Annee2, Mois1, Reste, I, Avant, Apres : INTEGER;
   MoisNum : ARRAY[1..12] OF INTEGER;

BEGIN
   RegC := 105;
   NEW (LitDate); {initialisation du pointeur}
   BDOS (RegC, ORD(LitDate)); {LitDate^ contient la date sous forme
                               de valeur numérique}
   BDOS (RegC, ADDR(Donnee)); {Donnee contient la date sous forme
                               de valeur numérique}

   Annee1 := LitDate^ DIV 1461;
   Reste  := LitDate^ MOD 1461;
   IF Reste <= 365 THEN
      BEGIN
         Annee2 := 0;
      END
   ELSE IF Reste <= 730 THEN
           BEGIN
              Annee2 := 1;
              Reste := Reste - 365;
           END
        ELSE IF Reste <= 1096 THEN
                BEGIN
                   Annee2 := 2;
                   Reste := Reste - 730;
                END
```

Figure 6.10 : *(suite)*

```
          ELSE
             BEGIN
               Annee2 := 3;
               Reste := Reste - 1096;
             END;
   MoisNum[1] := 31;
   MoisNum[2] := 28;
   MoisNum[3] := 31;
   MoisNum[4] := 30;
   MoisNum[5] := 31;
   MoisNum[6] := 30;
   MoisNum[7] := 31;
   MoisNum[8] := 31;
   MoisNum[9] := 30;
   MoisNum[10] := 31;
   MoisNum[11] := 30;
   MoisNum[12] := 31;
   IF Annee2 = 3 THEN MoisNum[2] := 29;
   I := 1;
   Avant := 0;
   Apres := MoisNum[1];
   WHILE I <= 12 DO
      BEGIN
         IF (Reste > Avant) AND (Reste <= Apres) THEN
            BEGIN
               STR((Reste - Avant),Jour);
               Mois1 := I;
               I := 12;
            END;
         Avant := Avant + MoisNum[I];
         Apres := Apres + MoisNum[I+1];
         I := I + 1;
      END;
   MoisNum[2] := 28;
   CASE (LitDate^ MOD 7) OF
```

Figure 6.10 : *(suite)*

```
      0 : JourS := 'Samedi';
      1 : JourS := 'Dimanche';
      2 : JourS := 'Lundi';
      3 : JourS := 'Mardi';
      4 : JourS := 'Mercredi';
      5 : JourS := 'Jeudi';
      6 : JourS := 'Vendredi';
   END;
   CASE Mois1 OF
      1 : Mois := 'Janvier';
      2 : Mois := 'Février';
      3 : Mois := 'Mars';
      4 : Mois := 'Avril';
      5 : Mois := 'Mai';
      6 : Mois := 'Juin';
      7 : Mois := 'Juillet';
      8 : Mois := 'Aout';
      9 : Mois := 'Septembre';
      10 : Mois := 'Octobre';
      11 : Mois := 'Novembre';
      12 : Mois := 'Décembre';
   END;
   STR (((1978+(Annee1*4))+Annee2),Annee);
   CDate := JourS + ' ' + Jour + ' ' + Mois + ' ' + Annee
END;

BEGIN
   WRITELN (CDate);
END.
```

Figure 6.10 *(suite)*

La fonction appelée DATE utilise la procédure Turbo BDOS (Fonction, I). La syntaxe de cette procédure requière pour l'exécution de la fonction 105 (lecture de la date et de l'heure système) deux paramètres : le numéro de la fonction BDOS et l'adresse d'une variable numérique entière à partir de laquelle le

système d'exploitation peut charger la date renvoyée par la fonction au Turbo. La date est représentée par une valeur entière stockée sur 16 bits, et pour laquelle le nombre 1 correspond au 1er janvier 1978. Attention, l'utilisation de certaines fonctions peut conduire à des résultats catastrophiques.

CP/M Plus permet d'accéder à 152 fonctions BDOS ; le système d'exploitation détermine laquelle des fonctions exécuter selon la valeur associée. Par exemple, la fonction 0 réinitialise le sytème, la fonction 1 lit un caractère entré au clavier, la fonction 2 affiche un caractère sur l'écran, la fonction 9 envoie une chaîne de caractères à l'imprimante, la fonction 19 supprime un fichier du disque, la fonction 23 renomme un fichier disque, etc.

Par conséquent, pour obtenir la date, on spécifie la fonction 105. Cette fonction peut renvoyer la date et l'heure ; dans ce cas, il faut lui spécifier une adresse mémoire à partir de laquelle elle écrira ces données :

```
Octets 0-1 : Champ date
Octet  2   : Champ heures
Octet  3   : Champ minutes
```

Les secondes sont retournées dans le registre A.

Pour lire la date en cours, il suffit de spécifier l'adresse d'une variable entière (2 octets) ; la date est renvoyée sous forme numérique, la valeur 1 correspond au 1er janvier 1978.

Le programme de la Figure 6.10 propose deux méthodes pour accéder à la fonction BDOS, passer l'adresse de la variable Donnee (ADDR(Donnee)) ou bien passer l'adresse contenue dans le pointeur LitDate (ORD(LitDate)) ; dans le premier cas, la date est stockée dans la variable numérique Donnee, dans le second cas, la date est stockée dans la variable LitDate^ pointée par LitDate :

```
BEGIN
   RegC := 105;
   NEW (LitDate); {initialisation du pointeur}
   BDOS (RegC, ORD(LitDate)); {LitDate^ contient la date sous
                                forme de valeur numérique}
```

```
BDOS (RegC, ADDR(Donnee)); {Donnee contient la date sous
                            forme de valeur numérique}
```

Dans le calendrier grégorien (promulgué par le pape Grégoire XIII en 1582), les années divisibles par quatre sont des années bissextiles (années de 366 jours dans lesquelles le mois de février contient 29 jours) à l'exception des années séculaires (ce sont les années pour lesquelles le millésime se termine par deux zéros ; 1800, 1900, 2000, etc.) dont le millésime n'est pas divisible par 400. Pour des raisons de simplicité dans le programme, d'une part, puis parce que la prochaine année séculaire dont le millésime n'est pas divisible par 400 est 2100, nous considérerons que le calendrier en cours est le calendrier julien (l'année julienne a une longueur moyenne de 365,25 jours alors qu'une année réelle a 365,2422 jours ; soit un excédent de 0,0078 jours).

Les années 1978, 1979 et 1981 ont 365 jours, l'année 1980 a 366 jours. Un cycle de quatre années (dont une année bissextile) possède 1461 jours, la troisième année (730 < Reste <= 1096) étant l'année bissextile :

```
Annee1 := LitDate^ DIV 1461;
Reste  := LitDate^ MOD 1461;
IF Reste <= 365 THEN
   BEGIN
      Annee2 := 0;
   END
ELSE IF Reste <= 730 THEN
        BEGIN
           Annee2 := 1;
           Reste := Reste - 365;
        END
     ELSE IF Reste <= 1096 THEN
             BEGIN
                Annee2 := 2;
                Reste := Reste - 730;
             END
          ELSE
             BEGIN
               Annee2 := 3;
```

```
            Reste := Reste - 1096;
          END;
```

L'année pourra ensuite être calculée par la fonction suivante :

```
STR (((1978+(Annee1*4))+Annee2),Annee);
```

Il faut maintenant déterminer le jour et le numéro du mois :

```
WHILE I <= 12 DO
   BEGIN
      IF (Reste > Avant) AND (Reste <= Apres) THEN
         BEGIN
            STR((Reste - Avant),Jour);
            Mois1 := I;
            I := 12;
         END;
      Avant := Avant + MoisNum[I];
      Apres := Apres + MoisNum[I+1];
      I := I + 1;
   END;
```

Si l'année est bissextile, il ne faut pas oublier d'attribuer 29 jours au mois de février (MoisNum[2]).

Ensuite, sachant que le 1er janvier 1978 était un dimanche, il faut déterminer le jour de la semaine :

```
CASE (LitDate^ MOD 7) OF
   0 : JourS := 'Samedi';
   1 : JourS := 'Dimanche';
   2 : JourS := 'Lundi';
   3 : JourS := 'Mardi';
   4 : JourS := 'Mercredi';
   5 : JourS := 'Jeudi';
   6 : JourS := 'Vendredi';
END;
```

Il ne reste plus enfin qu'à déterminer le nom du mois puis concaténer les résultats dans la chaîne CDate :

```
CASE Mois1 OF
   1 : Mois := 'Janvier';
   2 : Mois := 'Février';
   3 : Mois := 'Mars';
   4 : Mois := 'Avril';
   5 : Mois := 'Mai';
   6 : Mois := 'Juin';
   7 : Mois := 'Juillet';
   8 : Mois := 'Aout';
   9 : Mois := 'Septembre';
   10 : Mois := 'Octobre';
   11 : Mois := 'Novembre';
   12 : Mois := 'Décembre';
END;
STR (((1978+(Annee1*4))+Annee2),Annee);
CDate := JourS + ' ' + Jour + ' ' + Mois + ' ' + Annee
```

Le programme convertit les informations numériques renvoyées par le système en une chaîne pour faciliter leur insertion dans des rapports et leur affichage sur l'écran.

Le programme de la Figure 6.11 utilise encore la fonction BDOS numéro 105. Ce programme est une version modifiée du programme de tri de Shell de la Figure 4.1 ; il calcule et affiche le temps qui lui est nécessaire pour se dérouler et le temps dont il a besoin pour effectuer le tri de Shell. La fonction a un rôle de chronomètre intégré et a de nombreuses autres applications. L'appel de la fonction BDOS numéro 105 renvoie dans le registre A le temps contenu dans le champ secondes du chronomètre système. Le registre A est un registre 8 bits (un octet) et le nombre de secondes est renvoyé sous forme de 2 valeurs BCD codées sur quatre bits. Le nombre d'unités est obtenu en prenant le reste de la division entière du contenu du registre A par 16 (Secondes MOD 16). Le nombre de dizaines correspond au résultat de la division entière du contenu du registre A par 16 (Secondes DIV 16). Le tri de Shell appliqué à 500 nombres générés aléatoirement étant très inférieur à 60 secondes, il est inutile de lire le champ minutes du chronomètre système.

```
PROGRAM Tri_de_Shell_avec_Decompte_du_Temps_Ecoule;
{tri de Shell sur 500 nombres aléatoires}

CONST
   MaxEle = 500;

VAR
   Nums : ARRAY [1..500] OF INTEGER;
   Temp, I, J, Pass, Vide : INTEGER;
   E, B, T0, T1 : INTEGER;


PROCEDURE Compte (VAR TotalSecondes : INTEGER);
{lit le registre A contenant les secondes avec la fonction BDOS numéro 105}

TYPE
     Date = INTEGER;
     Adresse = ^Date;

VAR
   RegC : BYTE;
   LitDate : Adresse; {contient les valeurs renvoyées par le système}
   Secondes : INTEGER; {contient la valeur lue dans le registre A}

BEGIN
   RegC := 105;
   NEW (LitDate);
   Secondes := BDOS (RegC, ORD(LitDate)); {LitDate^ contient la date
                                           forme de valeur numérique}
   TotalSecondes := (Secondes DIV 16) * 10 + (Secondes MOD 16);
END;

PROCEDURE Entree_Gen;
BEGIN
```

Figure 6.11 : *Programme de la Figure 4.1 modifié pour afficher le temps nécessaire à la réalisation du tri*

```
   CLRSCR;
   WRITELN ('Début du programme de tri !');
   FOR I := 1 TO MaxEle DO
   BEGIN
      Nums[I] := RANDOM (5000);
   END;
END;

PROCEDURE Sortie;
BEGIN
   FOR I := 1 TO MaxEle DO
   BEGIN
      WRITE (Nums[I]);
      WRITE ('  ');
   END;
   WRITELN; WRITELN;
END;

PROCEDURE Interversion;
BEGIN
   Temp := Nums[J];
   Nums[J] := Nums[J + Vide];
   Nums[J + Vide] := Temp;
END;

PROCEDURE Tri;
BEGIN
   Vide := MaxEle DIV 2;
   WHILE Vide > 0 DO
   BEGIN
      FOR I := (Vide + 1) TO MaxEle DO
      BEGIN
         J := I - Vide;
         WHILE J > 0 DO
         BEGIN
```

Figure 6.11 *(suite)*

```
         IF Nums[J] > Nums[J + Vide] THEN
         BEGIN
            Interversion;
            J := J - Vide;
         END
         ELSE J := 0;
      END;                     {fin de while}
   END;
   Vide := Vide DIV 2;
  END;                         {tant que Vide >0}
END;                           {Tri}

BEGIN
   Compte (B);
   Entree_Gen;
   Compte (T0);
   Tri;
   Compte (T1);
   Sortie;
   Compte (E);
   IF E>B THEN E := E - B;
          ELSE E := (60 - B) + E;
   WRITE ('Il faut ', E, ' secondes pour générer, trier et afficher ');
   WRITELN (MaxEle, ' nombres.');
   WRITELN;
   IF T1>T0 THEN T1 := T1 - T0;
          ELSE E := (60 - T0) + T1;
   WRITE ('Il faut seulement ', T1, ' secondes pour le tri de ces ');
   WRITELN (MaxEle, ' nombres !!!');
END.
```

Figure 6.11 *(suite)*

Le programme charge dans la variable pointée par LitDate la valeur contenue dans le registre A puis la convertit en secondes.

ANNEXE
PROGRAMME DE GESTION DE COURRIER

Cette annexe contient l'ensemble du programme de gestion de courrier dont certaines parties ont été étudiées (sous une forme légèrement différente) dans le Chapitre 4. Il illustre la plupart des sujets traités dans cet ouvrage : structure modulaire, tri, recherche, tableaux, fichiers, enregistrements, interaction avec l'utilisateur, gestion d'erreurs, fichiers inclus pour les fonctions courantes et de nombreuses structures de contrôle. Plus important encore, il ne se contente pas d'illustrer ces caractéristiques dans des petits exemples factices mais les applique à un programme réel. Ce programme est utilisé tous les mois pour générer les étiquettes d'expédition d'une petite entreprise.

Bien qu'il ne soit pas très élégant, c'est un exemple de programme efficace. Même si un programme n'est jamais parfait (il y a toujours moyen de l'améliorer, de mieux appréhender les erreurs éventuelles), le but de la programmation est de résoudre les problèmes le jour même et non de prévoir un programme idéal pour le lendemain.

Ce programme n'emploie aucun algorithme complexe ni aucune logique obscure mais choisit toujours une solution simple et rationnelle plutôt qu'une solution qui offre une rapidité légèrement supérieure entraînant une complexité accrue.

Ce programme est documenté et contient des messages indiquant à l'utilisateur ce qu'il doit faire.

```
PROGRAM GestionDeCourrier;
{************************}
                           {utilise le fichier COMMUN.PAS}

CONST
   C = 'C'; {Contrôle de la tabulation, centrée, à gauche ou à droite}
   G = 'G';
   D = 'D';

TYPE
   EnregClient = RECORD
      Nom      : STRING[14];
      Prenom   : STRING[14];
      Rue      : STRING[28];
      Ville    : STRING[18];
      CodePost : STRING[5];
   END;
   LNom = STRING[14];
   Mots = STRING[64];
   NomEtCle = RECORD
      CodeEtNom : STRING[19]; {concatène le code postal et le nom
                               pour un tri dans un tri}
      Cle       : INTEGER;
   END;

VAR
   EntFich, SortFich               : FILE OF EnregClient;
   Client                          : EnregClient;
   Tri                             : ARRAY[1..100] OF NomEtCle;
   Tempo                           : NomEtCle;
   Correct                         : STRING[28];
   SourceFich, DestFich            : STRING[12];
   SurNom                          : STRING[14];
   Nom                             : LNom;
```

```
   EnregNum, Index, MaxEnreg, I, J : INTEGER;
   LignesFaite, Option             : INTEGER;
   ToutFini, Interv                : BOOLEAN;
   Encore                          : CHAR;
   Minimum, Maximum, Cherche       : INTEGER;
   Efface, Trouve, FaitAutre       : BOOLEAN;


{$I COMMUN.PAS}    {contient les procédures Tab, CDate et Sin_A}
```

```
PROCEDURE AfficheMenu;
{-------------------}
{tous les programmes de base sont sélectionnés à partir de ce menu
après chaque exécution, le programme revient à celui-ci}

VAR
  Aujourdhui : STRING[40];

BEGIN
   Aujourdhui := CDate;
   CLRSCR;
   GOTOXY (15,2); WRITE ('Programme de gestion de courrier, Turbo Pascal
');
   WRITELN ('-- ', Aujourdhui, ' --');
   GOTOXY (35,5); WRITELN ('Menu de commandes :');
   GOTOXY (30,7); WRITELN ('1. Introduction de nouveaux noms');
   GOTOXY (30,9); WRITELN ('2. Lire ou modifier des données');
   GOTOXY (30,11); WRITELN ('3. Tri alphabétique ou numérique');
   GOTOXY (30,13); WRITELN ('4. Affichage d une liste sur l écran');
   GOTOXY (30,15); WRITELN ('5. Création d étiquettes');
   GOTOXY (30,17); WRITELN ('6. Sortie de données');
   GOTOXY (30,19); WRITELN ('7. Copie de fichier ou fusion de noms');
   GOTOXY (30,21); WRITELN ('8. Fin du programme');
   GOTOXY (25,23); WRITELN ('Choisissez un nombre !');
   GOTOXY (55,23);
END;


PROCEDURE GestionD_Erreur;
{======================}
{peut être plus élaboré ; il pourrait revenir à l'affichage du menu
si la valeur entrée n'est pas valide}

BEGIN
   WRITELN ('Condition d erreur');
   DELAY (700); {temps nécessaire à la lecture du message}
END;
```

```
PROCEDURE CopieFichier;
{--------------------}
{copie le contenu d'un fichier dans un autre et applique un filtre sur
les enregistrements qui ont été marqués pour suppression. Cette procédure
est appelée par une autre et ne doit donc pas contenir d'instructions
d'initialisation comme RESET, REWRITE et ASSIGN}

BEGIN
   WHILE NOT EOF (EntFich) DO
      BEGIN
         READ (EntFich, Client);
         IF Client.Prenom <> '@' THEN WRITE (SortFich, Client);
      END;
      CLOSE (EntFich);
      CLOSE (SortFich);
END;
```

```
PROCEDURE NouveauxNoms;
{================================}
{choix #1.
cette procédure place les nouveaux noms dans un fichier temporaire
pour permettre une correction plus facile. Lorsque le fichier est correct,
il est ajouté à la liste principale}

VAR
   ToutFait : BOOLEAN;

BEGIN
   CLRSCR;
   WRITELN ('Nom du fichier destiné à recevoir les données temporaires :');
   WRITELN ('(pour NOUVNOM.TMP, taper Return)'); {nom du fichier par
défaut}
   READLN (DestFich);
   IF DestFich = '' THEN ASSIGN (SortFich, 'NOUVNOM.TMP')
   ELSE ASSIGN (SortFich, DestFich);

{cette procédure suppose que le fichier temporaire n'existe pas déjà et
le message d'erreur ("File does not exist") n'est pas généré grâce à la
directive de compilation qui suit la zone de remarques ; la fonction
IORESULT retourne la valeur 1. Si le fichier existe, le déroulement n'est
pas affecté ; la fonction IORESULT retourne la valeur 0}

   {$I-} {désactivation de la gestion des erreurs d'entrée/sortie}
   RESET (SortFich); {tentative de placer le pointeur au début du fichier}
   {$I+} {activation de la gestion des erreurs d'entrée/sortie}
   IF IORESULT <> 0 THEN REWRITE (SortFich); {création du fichier si
                                              nécessaire}

   MaxEnreg := FILESIZE (SortFich); {détermine le numéro du dernier
                                     enregistrement}
   SEEK (SortFich, MaxEnreg);       {place le pointeur à la fin du fichier}
   ToutFait := FALSE;               {initialise la boucle pour un nouveau
                                     nom}
   WHILE NOT ToutFait DO
      BEGIN
```

```
        WITH Client DO
        BEGIN
           WRITELN ('Taper un astérisque (*) pour mettre fin à la
saisie.');
           WRITELN;
           WRITELN ('Nom du client :');
           READLN (Nom);
           IF Nom <> '*' THEN
              BEGIN
                 WRITELN ('Prénom du client :');
                 READLN (Prenom);
                 WRITELN ('Rue:');
                 READLN (Rue);
                 WRITELN ('Ville :');
                 READLN (Ville);
                 WRITELN ('Code postal :');
                 READLN (CodePost);
                 WRITE (SortFich, Client);
              END
           ELSE ToutFait := TRUE;
        END; {fin du traitement d'un enregistrement}
     END; {fin de boucle si ToutFait est vrai (TRUE)}
  CLOSE (SortFich);
END;
```

```
PROCEDURE EntreeFichierExt;
{-----------------------}
{liée au choix #3 du menu.
Initialise le fichier en vue d'un tri alphabétique ou suivant le code
postal. Cette procédure charge un tableau à deux dimensions constitué
de clés (concaténation du code postal et du prénom) et de numéros
d'enregistrement.}

VAR
   AlphaSeul : BOOLEAN;
   AlphaCode : CHAR;
   Nombre : STRING[5];

BEGIN
   CLRSCR;
   AlphaSeul := FALSE;
   WRITELN ('Fichier à trier :');
   READLN (SourceFich);
   ASSIGN (EntFich, SourceFich);
   WRITELN ('Tri alphabétique (A) ou par code postal (C) ?');
   READLN (AlphaCode);
   AlphaCode := UPCASE (AlphaCode);
   IF AlphaCode = 'A' THEN AlphaSeul := TRUE;
   WRITELN ('Fichier destiné à recevoir les données triées :');
   READLN (DestFich);
   ASSIGN (SortFich, DestFich);
   RESET (EntFich);
   EnregNum := 0;
   WHILE NOT EOF (EntFich) DO {chargement du tableau}
      BEGIN
         READ (Entfich, Client);
         EnregNum := EnregNum + 1;
         Nombre := Client.CodePost;

{cette méthode permet d'exécuter un tri dans un tri. En concaténant un
code postal et un nom, le programme place la valeur 06457Wagner avant
94211Bach et 07011Dupond avant 07011Durant. Pour un tri alphabétique
```

seul, la valeur du code postal est mise à zéro ; ainsi 0Bach est placé avant 0Wagner.}

```
         IF AlphaSeul THEN Nombre :='0';
         Tri[EnregNum].CodeEtNom := Nombre + Client.Nom;
         Tri[EnregNum].Cle := EnregNum;
      END;
   MaxEnreg := EnregNum;
   CLOSE (EntFich);
END;
```

```
PROCEDURE EcritNouvFich;
{---------------------}
{liée au choix #3 du menu.
Lorsque le tableau de clés et de numéros d'enregistrements a été trié, les
enregistrements sont lus dans leur ordre correct puis écrits dans le
fichier de sortie. Pour de gros fichiers, le traitement peut être assez
long ; il est conseillé d'utiliser des disques virtuels ou de réécrire la
procédure en utilisant des pointeurs.}

BEGIN
   ASSIGN (EntFich, SourceFich);
   RESET (EntFich);
   ASSIGN (SortFich, DestFich);
   REWRITE (SortFich);
   FOR I := 1 TO MaxEnreg DO
      BEGIN
         RESET (EntFich);
         SEEK (EntFich, Tri[I].Cle-1);
         READ (EntFich, Client); {lit l'enregistrement complet}
         WRITE (SortFich, Client);
      END;
   CLOSE (EntFich);
   CLOSE (SortFich);
END;


PROCEDURE Fini;
{------------}
{liée au choix #3 du menu.
cette procédure avertit l'utilisateur que le tri est terminé en affichant
un message puis en traçant la sinusoïde}
BEGIN
   CLRSCR;
   WRITELN ('Le fichier ', DestFich, ' est créé !');
   WRITELN ('Il contient ', MaxEnreg, ' enregistrements.');
   Sin_A {Dessine la sinusoïde pour distraire l'utilisateur}
END;
```

```
PROCEDURE Intervert;
{-----------------}
{liée au choix #3 du menu.
intervertit les éléments d'un tableau. Elle a été séparée de la procédure
de tri simplement pour la rendre plus claire}

BEGIN
   Tempo      := Tri[J];
   Tri[J]     := Tri[J + 1];
   Tri[J + 1] := Tempo;
   Interv     := TRUE;
END;
```

```
PROCEDURE TriProg;
{---------------}
{liée au choix #3 du menu. Tri de Shell appliqué au tableau
d'enregistrements de codes postaux et noms}
VAR
   Passe, Vide : INTEGER;
BEGIN
   Vide := MaxEnreg DIV 2;
   WHILE Vide > 0 DO
      BEGIN
         FOR I := (Vide + 1) TO MaxEnreg DO
            BEGIN
               J := I - Vide;
               WHILE J > 0 DO
                  BEGIN
                     IF Tri[J].CodeEtNom > Tri[J + Vide].CodeEtNom THEN
                        BEGIN
                           Intervert;
                           J := J - Vide;
                        END
                     ELSE J := 0;
                  END;
            END;
            Vide := Vide DIV 2;
         END;
END;

PROCEDURE ListeAlphabetique;
{==========================}
{choix #3.
Comme cette opération est complexe, elle a été séparée en
plusieurs procédures simples}
BEGIN
   EntreeFichierExt;
   TriProg;
   EcritNouvFich;
   Fini;
END;
```

```
PROCEDURE CorrigeNom;
{------------------}
{appelée par la procédure TestNom, elle permet à l'utilisateur de corriger
un enregistrement dans un fichier. En remplaçant le prénom par le caractère
"@", un enregistrement peut être marqué pour effacement.}

BEGIN
   WITH Client DO
      BEGIN
         WRITELN (Prenom);
         WRITELN ('Remplacer par :');
         READLN (Correct);
         IF Correct <> '' THEN Prenom := Correct;
         WRITELN (Nom);
         WRITELN ('Remplacer par :');
         READLN (Correct);
         IF Correct <> '' THEN Nom := Correct;
         WRITELN (Rue);
         WRITELN ('Remplacer par :');
         READLN (Correct);
         IF Correct <> '' THEN Rue := Correct;
         WRITELN (Ville);
         WRITELN ('Remplacer par :');
         READLN (Correct);
         IF Correct <> '' THEN Ville := Correct;
         WRITELN (CodePost);
         WRITELN ('Remplacer par :');
         READLN (Correct);
         IF Correct <> '' THEN Codepost := Correct;
         SEEK (EntFich, Cherche);
         WRITE (EntFich, Client);
         WRITELN ('Correction d un autre nom ? O/N');
         READ (KBD, Encore);
         Encore := UPCASE (Encore);
         IF Encore = 'O' THEN
            BEGIN
               FaitAutre := TRUE;
               Trouve := FALSE;
```

```
                CLRSCR;
                WRITELN ('Nom à modifier :');
                READLN (SurNom);
                Minimum := 0;
                Maximum := MaxEnreg;
             END
          ELSE FaitAutre := FALSE;
      END;
END;
```

```
PROCEDURE EffaceEntree;
{--------------------}
{partie de la procédure TestNom, remplace le prénom par "@" et propose
d'effacer un autre enregistrement. Lorsque les noms à supprimer ont été
marqués, la procédure CompacteFich est appelée pour "nettoyer" le fichier}

BEGIN
   Client.Prenom := '@';
   SEEK (EntFich, Cherche);
   Efface := TRUE;
   CLRSCR;
   WRITELN ('Suppression d un autre nom (O/N) ?');
   READ (KBD, Encore);
   Encore := UPCASE (Encore);
   IF Encore = 'O' THEN
      BEGIN
         FaitAutre := TRUE;
         Trouve := FALSE;
         CLRSCR;
         WRITELN ('Nom à visualiser :');
         READLN (SurNom);
         Minimum := 0;
         Maximum := MaxEnreg;
      END
   ELSE FaitAutre := FALSE;
END;


PROCEDURE CompacteFich;
{--------------------}
{utilisée par la procédure TestNom, elle "filtre" les enregistrement
marqués pour suppression. Elle fait appel aux commandes RENAME et ERASE}

BEGIN
   WRITELN ('La procédure de suppression peut durer quelques minutes !');
   RENAME (EntFich, '@@.@@@');
   RESET (EntFich);
```

```
   ASSIGN (SortFich, 'TEST.TMP');
   REWRITE (SortFich);
   CopieFichier;
   RENAME (SortFich, SourceFich);
   ERASE (EntFich);
END;
```

```
PROCEDURE TestNom;
{---------------}
{associée à la procédure TrouveUnNom, elle appelle d'autres procédures pour
modifier et supprimer des entrées}

VAR
   DerRech : INTEGER;

BEGIN
   Efface := FALSE;
   WHILE NOT Trouve DO
      BEGIN
         Cherche := (Minimum + Maximum) DIV 2;
         IF Cherche = DerRech THEN Minimum := MaxInt;
         DerRech := Cherche;
         SEEK (EntFich, Cherche);
         READ (EntFich, Client);
         IF SurNom = Client.Nom THEN
            BEGIN
               Trouve := TRUE;
               CLRSCR;
               GOTOXY (1,12);
               WRITELN (Client.Prenom, ' ', Client.Nom);
               WRITELN (Client.Rue);
               WRITELN (Client.Ville, ' ', Client.CodePost);
               WRITELN;
               WRITE ('(L)it un autre nom, (E)fface l entrée, ');
               WRITELN ('(C)orrige l entrée, (R)evient au menu.');
               READ (KBD, Encore);
               Encore := UPCASE (Encore);
               CASE Encore OF
                  'L':
                     BEGIN
                        FaitAutre := TRUE;
                        Trouve := FALSE;
                        CLRSCR;
                        WRITELN ('Nom à trouver :');
                        READLN (SurNom);
```

```
                    Minimum := 0;
                    Maximum := MaxEnreg;
                  END;
                'E': EffaceEntree;
                'C': CorrigeNom;
                'R': FaitAutre := FALSE;
            END;
        END;
        IF Minimum < Maximum THEN {si la recherche est valide}
          BEGIN
            IF SurNom > Client.Nom THEN
              BEGIN
                Minimum := Cherche;
              END;
            IF SurNom < Client.Nom THEN
              BEGIN
                Maximum := Cherche;
              END;
          END
        ELSE
          BEGIN
            WRITELN ('Je ne trouve pas ce nom !');
            WRITELN;
            FaitAutre := TRUE;
            Trouve := FALSE;
            WRITELN ('Nom à rechercher :');
            READLN (SurNom);
            Minimum := 0;
            Maximum := MaxEnreg;
          END;
      END;
  CLOSE (EntFich);
  IF Efface THEN CompacteFich;
END;
```

```
PROCEDURE LitUnNom;
{=================}
{choix #2.
Initialisation pour l'exécution d'un tri ; cette procédure utilise les
enregistrements d'un fichier externe ainsi que plusieurs procédures}

BEGIN
   CLRSCR;
   WRITELN ('Nom du fichier à scruter :');
   READLN (SourceFich);
   ASSIGN (EntFich, SourceFich);
   RESET (EntFich);
   Trouve := FALSE;
   MaxEnreg := FILESIZE (EntFich); {cherche le maximum}
   WRITELN ('Taille du fichier --> ', MaxEnreg);
   FaitAutre := TRUE;
   WHILE FaitAutre DO
      BEGIN
         Minimum := 0;
         Maximum := MaxEnreg;
         WRITELN ('Nom à chercher :');
         READLN (SurNom);
         TestNom;
      END;
END;
```

```
PROCEDURE ListeSurEcran;
{====================}
{choix #4.
elle fait appel à la procédure Tab pour rendre plus attractif
l'affichage sur l'écran}

VAR
   L1, L2, L3, L4 : STRING[40];

BEGIN
   CLRSCR;
   WRITELN ('Nom du fichier à afficher :');
   READLN (DestFich);
   ASSIGN (SortFich, DestFich);
   RESET (SortFich);
   WHILE NOT EOF (SortFich) DO
      BEGIN
         READ (SortFich, Client);
         WITH Client DO
            BEGIN
               L1 := Client.Prenom + '  ' + Client.Nom;
               Tab (L1, 28, G);
               L2 := Client.Rue;
               Tab (L2, 28, G);
               L3 := Client.Ville;
               Tab (L3, 18, G);
               L4 := Client.CodePost;
               Tab (L4, 5, G);
               WRITELN (L1, L2, L3, L4);
            END;
      END;
   CLOSE (SortFich);
   WRITELN ('Taper Return pour revenir au menu.');
   READ (Encore);
END;
```

```
PROCEDURE Etiquette;
{=================}
{choix #5.
cette procédure génère des étiquettes ; celles-ci sont envoyées sur
l'imprimante}

BEGIN
   CLRSCR;
   WRITELN ('Nom du fichier contenant les noms :');
   READLN (DestFich);
   ASSIGN (SortFich, DestFich);
   RESET (SortFich);

{les codes envoyés à l'imprimante sont ceux de la NEC Spinwriter utilisant
des étiquettes de 1 pouce. Dans le cas d'un autre type d'imprimante, il
faut modifier l'initialisation}

   WRITE (LST, #27, #67, #6); {6 lignes pour chaque étiquette}
   WHILE NOT EOF (SortFich) DO
      BEGIN
         READ (SortFich, Client);
         WITH Client DO
            BEGIN
               WRITELN (LST);
               WRITELN (LST, Prenom, '  ', Nom);
               WRITELN (LST, Rue);
               WRITELN (LST, Ville, '  ', CodePost);
            END;
         WRITELN (LST, #12); {envoi d'un Form Feed}
      END;
END;
```

```
PROCEDURE SortieDonnees;
{=====================}
{choix #6.
cette procédure sort les données sur un périphérique quelconque, elle
permet de définir des marges, des en-têtes et autres structures liées
à la présentation}

VAR
   Spool : TEXT;
   Affiche : BOOLEAN;
   Uneligne : STRING[132];
   L1, L2, L3, L4 : STRING[40];

BEGIN
   CLRSCR;
   WRITELN ('Nom du fichier contenant les noms :');
   READLN (SourceFich);
   ASSIGN (EntFich, SourceFich);
   RESET (EntFich);
   WRITELN ('Taper un nom de fichier si les résultats doivent');
   WRITELN ('être stockés pour une impression ultérieure.');
   WRITELN ('Sinon, pour une impression immédiate, taper Return');
   READLN (DestFich);
   IF DestFich <> '' THEN
      BEGIN
         ASSIGN (Spool, DestFich);
         REWRITE (Spool);
         Affiche := TRUE;
      END;

{ces codes d'initialisation sont spécifiques de la NEC Spinwriter
ils doivent être modifiés pour une autre imprimante}

   WRITE (Spool, #27, #67, #66); {longueur de page, 66 lignes}
   WRITE (Spool, #27, '$A', #27, #85, #1, #27, #87, #82);
   L1 := 'GESTION DE COURRIER TURBO PASCAL';
   Tab (L1, 80, C);
   WRITELN (Spool, L1);
```

```
L1 := CDate;
Tab (L1, 80, C);
WRITELN (Spool, L1);
WRITELN (Spool, #10, #10, #10); {sauts de ligne}
LignesFaite := 7;
WHILE NOT EOF (EntFich) DO
   BEGIN
      IF LignesFaite < 59 THEN {test page remplie}
         BEGIN
            WITH Client DO
               BEGIN
                  READ (EntFich, Client);
                  {mise en page des résultats}
                  L1 := Prenom + '  ' + Nom;
                  Tab (L1, 30, G);
                  L2 := Rue;
                  Tab (L2, 30, G);
                  L3 := Ville;
                  Tab (L3, 20, G);
                  L4 := CodePost;
                  Tab (L4, 5, G);
                  WRITELN (Spool, L1, L2, L3, L4);
                  LignesFaite := LignesFaite + 1;
               END;
         END
      ELSE
         BEGIN {génère un nouvel en-tête}
            LignesFaite := 8;
            WRITELN (Spool, #12, #10, #10); {changement de page et 2
                                              sauts de ligne}
            L1 := 'GESTION DE COURRIER TURBO PASCAL';
            Tab (L1, 80, C);
            WRITELN (Spool, L1);
            L1 := CDate;
            Tab (L1, 80, C);
            WRITELN (Spool, L1);
            WRITELN (Spool, #10, #10, #10); {sauts de ligne}
```

```
        END; {fin de else}
    END; {fin de while}
  CLOSE (EntFich);
  CLOSE (Spool);
  IF Affiche THEN {s'il n'y a pas de redirection des résultats}
    BEGIN
      RESET (Spool);
      WHILE NOT EOF (Spool) DO
        BEGIN
          READLN (Spool, UneLigne);
          WRITELN (LST, UneLigne);
        END;
    END;
END;
```

```
PROCEDURE CopieOuFusion;
{=====================}
{choix #7.
cette procédure permet d'ajouter la liste de nouveaux noms à la liste
principale ou de faire la copie d'un fichier ; elle utilise la procédure
CopieFichier}

BEGIN
   CLRSCR;
   WRITE ('(C)opier un fichier ou (F)usionner les nouveaux noms avec ');
   WRITELN ('la liste principale.');
   READ (KBD, Encore);
   Encore := UPCASE (Encore);
   CASE Encore OF
      'C' :
         BEGIN
            WRITELN ('Nom du fichier source :');
            READLN (SourceFich);
            WRITELN ('Nom du fichier destination :');
            READLN (DestFich);
            ASSIGN (EntFich, SourceFich);
            RESET (EntFich);
            ASSIGN (SortFich, DestFich);
            REWRITE (Sortfich);
            CopieFichier;
         END;
      'F' :
         BEGIN
            WRITELN ('Nom du fichier contenant les nouveaux noms :');
            READLN (SourceFich);
            WRITELN ('Nom du fichier principal :');
            READLN (DestFich);
            ASSIGN (EntFich, SourceFich);
            RESET (EntFich);
            ASSIGN (SortFich, DestFich);
            RESET (SortFich);
            MaxEnreg := FILESIZE (SortFich); {n  du dernier enregistrement}
            SEEK (SortFich, MaxEnreg); {pointe la fin du fichier}
```

```
            WHILE NOT EOF (EntFich) DO
               BEGIN
                  READ (EntFich, Client);
                  WRITE (SortFich, Client);
               END;
            CLOSE (EntFich);
            CLOSE (SortFich);
         END;
      ELSE CopieOuFusion;
   END;
END;
```

```
PROCEDURE InterpreteChoix;
{----------------------}

BEGIN
   READ (KBD, Encore);
   Option := ORD (Encore) - 48;
   CASE Option OF
      1 : NouveauxNoms;
      2 : LitUnNom;
      3 : ListeAlphabetique;
      4 : ListeSurEcran;
      5 : Etiquette;
      6 : SortieDonnees;
      7 : CopieOuFusion;
      8 : ToutFini := True;
      ELSE GestionD_Erreur;
   END;
END;


{****************************************}

BEGIN
{l'exécution du programme commence ici}
   ToutFini := FALSE;
   WHILE NOT ToutFini DO
      BEGIN
         AfficheMenu;
         InterpreteChoix;
      END;
END.

{***************************************}
```

LE FICHIER COMMUN.PAS

```
{les trois procédures suivantes font partie du fichier COMMUN.PAS}

{La première fonction appelée CDate lit la date à partir du système
 d'exploitation et la convertit en une chaîne de caractères}

TYPE
   ChaineDate = STRING[40];

FUNCTION CDate : ChaineDate;  {retourne la date décodée}

TYPE
     Date = INTEGER;
     Adresse = ^Date;

VAR
   RegC : BYTE;
   LitDate : Adresse; {contient les valeurs renvoyées par le système}
   Donnee : INTEGER; {contient les valeurs renvoyées par le système}
   Mois : STRING[8];
   Jour : STRING[2];
   Annee : STRING[4];
   JourS : STRING[8];
   Annee1, Annee2, Mois1, Reste, I, Avant, Apres : INTEGER;
   MoisNum : ARRAY[1..12] OF INTEGER;

BEGIN
   RegC := 105;
   NEW (LitDate); {initialisation du pointeur}
   BDOS (RegC, ORD(LitDate)); {LitDate^ contient la date sous forme
                               de valeur numérique}
   BDOS (RegC, ADDR(Donnee)); {Donnee contient la date sous forme
                               de valeur numérique}

   Annee1 := LitDate^ DIV 1461;
   Reste  := LitDate^ MOD 1461;
   IF Reste <= 365 THEN
      BEGIN
         Annee2 := 0;
```

```
   END
ELSE IF Reste <= 730 THEN
        BEGIN
           Annee2 := 1;
           Reste := Reste - 365;
        END
     ELSE IF Reste <= 1096 THEN
             BEGIN
                Annee2 := 2;
                Reste := Reste - 730;
             END
          ELSE
             BEGIN
               Annee2 := 3;
               Reste := Reste - 1096;
             END;
MoisNum[1] := 31;
MoisNum[2] := 28;
MoisNum[3] := 31;
MoisNum[4] := 30;
MoisNum[5] := 31;
MoisNum[6] := 30;
MoisNum[7] := 31;
MoisNum[8] := 31;
MoisNum[9] := 30;
MoisNum[10] := 31;
MoisNum[11] := 30;
MoisNum[12] := 31;
IF Annee2 = 3 THEN MoisNum[2] := 29;
I := 1;
Avant := 0;
Apres := MoisNum[1];
WHILE I <= 12 DO
   BEGIN
      IF (Reste > Avant) AND (Reste <= Apres) THEN
         BEGIN
            STR((Reste - Avant),Jour);
            Mois1 := I;
```

```
            I := 12;
          END;
        Avant := Avant + MoisNum[I];
        Apres := Apres + MoisNum[I+1];
        I := I + 1;
      END;
    MoisNum[2] := 28;
    CASE (LitDate^ MOD 7) OF
      0 : JourS := 'Samedi';
      1 : JourS := 'Dimanche';
      2 : JourS := 'Lundi';
      3 : JourS := 'Mardi';
      4 : JourS := 'Mercredi';
      5 : JourS := 'Jeudi';
      6 : JourS := 'Vendredi';
    END;
    CASE Mois1 OF
      1 : Mois := 'Janvier';
      2 : Mois := 'Février';
      3 : Mois := 'Mars';
      4 : Mois := 'Avril';
      5 : Mois := 'Mai';
      6 : Mois := 'Juin';
      7 : Mois := 'Juillet';
      8 : Mois := 'Aout';
      9 : Mois := 'Septembre';
      10 : Mois := 'Octobre';
      11 : Mois := 'Novembre';
      12 : Mois := 'Décembre';
    END;
    STR (((1978+(Annee1*4))+Annee2),Annee);
    CDate := JourS + ' ' + Jour + ' ' + Mois + ' ' + Annee
  END;
```

```
{La procédure Tab contrôle la position horizontale d'une chaîne sur
 l'écran ou sur une imprimante}

PROCEDURE Tab (VAR Phrase : EntChaine; Champ : INTEGER; Pos : CHAR);
{-----------  c'est la seule ligne modifiée}

VAR
   EspacesG, EspacesD : STRING[80];
   I, EspacesCoteG, EspacesCoteD, EspacesSurLigne : INTEGER;

BEGIN
   EspacesG := '';
   EspacesD := '';
   EspacesSurLigne := Champ - LENGTH (Phrase);

   IF (Pos = 'C') OR (Pos = 'c') THEN  {centré}
   BEGIN
      EspacesCoteG := EspacesSurLigne DIV 2;
      EspacesCoteD := (EspacesSurLigne DIV 2) + (EspacesSurLigne MOD 2);
      FOR I :=  1 TO EspacesCoteG DO EspacesG := EspacesG + '_';
      FOR I :=  1 TO EspacesCoteD DO EspacesD := EspacesD + '_';
   END;

   IF (Pos = 'D') OR (Pos = 'd') THEN  {cadré à droite}
   BEGIN
      EspacesCoteG := EspacesSurLigne;
      FOR I :=  1 TO EspacesCoteG DO EspacesG := EspacesG + '_';
   END;

   IF (Pos = 'G') OR (Pos = 'g') THEN  {cadré à gauche}
   BEGIN
      EspacesCoteD := EspacesSurLigne;
      FOR I :=  1 TO EspacesCoteD DO EspacesD := EspacesD + '_';
   END;

   Phrase := EspacesG + Phrase + EspacesD;
END;
```

```
{La troisième procédure, nécessite la bibliothèque Turbo Graphix Toolbox ;
elle dessine sur l'écran une sinusoïde amortie}


PROCEDURE Sin_A;


{$I TYPEDEF.SYS}            (* ces fichiers doivent etre
{$I GRAPHIX.SYS}               inseres dans cet ordre *)
{$I KERNEL.SYS}
{$I KERNEL1.SYS}


CONST
   Pi = 3.1416;


FUNCTION Y(X : REAL) : REAL;
BEGIN
   Y := (EXP( 0.02 * X) * SIN (X) * 100) + 99;
END;


PROCEDURE Trace_axes;
BEGIN
   DRAWSTRAIGHT (19,619,100);
   DRAWLINECLIPPED (319,9,319,189);
END;


PROCEDURE Trace_gradu;
VAR
   I : INTEGER;

BEGIN
   I := 19;
   WHILE I <= 619 DO
      BEGIN
```

```
         DRAWLINECLIPPED (I,97,I,101);
         I := I + 20;
      END;
   I := 9;
   WHILE I <= 199 DO
      BEGIN
         DRAWSTRAIGHT (316,312,I);
         I := I + 20;
      END;
END;


PROCEDURE Genere_fonction;
VAR
   X0, Y0 : INTEGER;
   X, I   : REAL;

BEGIN
   I := 0;
   X := 0;
   X0 := 19;
   Y0 := 100;
   WHILE I <= (30 * Pi) DO
      BEGIN
         X := X + 1;
         DRAWLINECLIPPED (X0,Y0,ROUND(X + 19),ROUND(Y(I));
         X0 := ROUND (X + 19);
         Y0 := ROUND (Y(I));
         I := I + (5 * Pi / 100);
      END;
END;


PROCEDURE Trace_fig;
BEGIN
   CLEARSCREEN;
   DRAWBORDER;
   Trace_axes;
```

```
   Trace_gradu;
   Genere_fonction;
END;


BEGIN
   INITGRAPHIC;                 {initialise le systeme graphique}
   Trace_fig;
   REPEAT UNTIL KEYPRESSED;     {attend la frappe d'une touche}
   LEAVEGRAPHIC;                {quitte le systeme graphique}
END;
```

POUR UN CATALOGUE COMPLET DE NOS PUBLICATIONS

FRANCE
6-8, Impasse du Curé
75881 PARIS CEDEX 18
Tél. : (1) 42.03.95.95
Télex : 211801

U.S.A.
2344 Sixth Street
Berkeley, CA 94710
Tel. : (415) 848.8233
Telex : 336311

ALLEMAGNE
Vogelsanger. WEG 111
4000 Düsseldorf 30
Postfach N° 30.09.61
Tel. : (0211) 61 80 2-0
Telex : 08588163



Paris • Berkeley • Düsseldorf



Achevé d'imprimer par l'imprimerie PRAXIE, 18-20, bd de Ménilmontant - 75020 PARIS

Dépôt légal 4° trimestre 1986

Imprimeur N° 1002

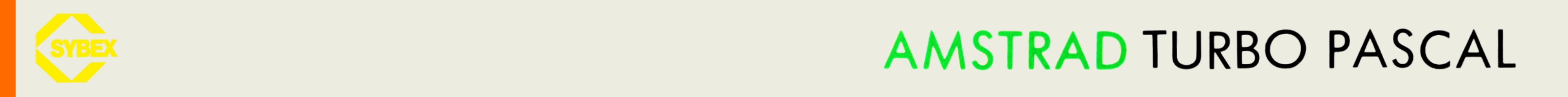
Ce livre est destiné à tous les utilisateurs du compilateur Turbo Pascal de Borland sur Amstrad, qu'ils soient programmeurs expérimentés ou débutants. Toutes les caractéristiques et particularités de ce compilateur sont présentées et étudiées en détail. Chaque sujet abordé est illustré par un ou plusieurs exemples de programmes. Cet ouvrage couvre non seulement les éléments propres au langage, mais également l'interaction avec le système et, en particulier :

- structure des programmes ;
- structure des données ;
- graphisme ;
- sons ;
- utilisation des routines système ;
- utilisation des options du compilateur et du linker.

L'AUTEUR

Douglas Stivison est diplômé de l'Université de l'état de New York et consultant en informatique.




SYBEX
AMSTRAD TURBO PASCAL

